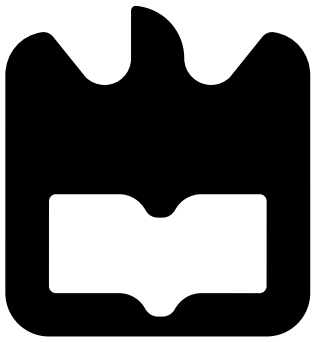
Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2018

João Pedro Carvalho Pires

Ferramentas de Desenvolvimento para Agentes de DOTA 2



João Pedro Carvalho Pires

Ferramentas de Desenvolvimento para Agentes de DOTA 2

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica de Artur José Carneiro Pereira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Professor Doutor Luís Filipe de Seabra Lopes
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Artur José Carneiro Pereira
Professor Auxiliar da Universidade de Aveiro (orientador)

Professor Doutor Rosaldo José Fernandes Rossetti
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

agradecimentos / acknowledgements

Agradeço a todos os que me acompanharam durante o ano e meio de desenvolvimento desta tese, nomeadamente os meus pais, o meu irmão e os meus amigos. Aproveito também para agradecer ao Professor Artur, Professor Nuno e ao David Simões, pela orienteção fundamental que me deram, sempre que precisei.

palavras-chave

DOTA 2, Videojogos, Machine Learning, Inteligência Artificial, Ferramenta de *Software*, Ferramenta de Código Aberto

resumo

Atualmente, são desenvolvidos algoritmos de ML (*Machine Learning*) para qualquer ramo da ciência, de forma a resolver problemas dos mais variados géneros. Um ótimo exemplo deste tipo de problema é a área dos videojogos. Apesar de não ser recente, a aplicação de ML para a criação de Inteligências Artificiais (IAs) em videojogos, tem-se tornado num tema cada vez mais desenvolvido, levando a um maior interesse nesta área. Isto deve-se ao crescimento da complexidade de certo tipo de videojogos, como é o exemplo do Defense Of The Ancients 2 (DOTA 2). Atualmente, este videojogo é um dos mais complexos e difícies de jogar, quer seja pela quantidade de informação que um jogador tem de processar ao longo de uma partida, quer seja pela quantidade de decisões que podem ser tomadas em determinado instante da mesma. A dificuldade do DOTA 2 levou a que a criação de IAs para agentes deste jogo se tornasse num trabalho muito interessante e desafiante.

Infelizmente, apesar de existirem implementações muito conhecidas e indiscutivelmente eficazes de IAs para agentes de DOTA 2, não existem ferramentas de *software* de código livre que possibilitem, a um utilizador comum, efetuar este género de implementação. É devido a este facto que se decidiu criar uma *framework*, nesta tese, de uso simples, com o objetivo de permitir o desenvolvimento de IAs para agentes de DOTA 2 que facilite o processo de aprendizagem e teste dos agentes implementados, recorrendo à automatização de criação de ambientes através da *Graphical User Interface* (GUI) deste videojogo. Esta dissertação irá focar todos os aspetos relevantes desta interface entre o jogo e um ambiente externo de programação, utilizando a linguagem de programação Python, nomeadamente a Arquitetura do Sistema, onde são definidas todas as características fundamentais da solução apresentada, e a Implementação, onde se encontra detalhado o processo de desenvolvimento deste sistema.

keywords

DOTA 2, Video Games, Machine Learning, Artificial Intelligence, Software Framework, Open-source Framework

abstract

Nowadays, ML (Machine Learning) algorithms are developed to solve problems of various genres, in every field of science. A great example of this kind of problem is the field of video gaming. Although it is not a recent topic, the use of ML to create AI (Artificial Intelligence) in video games is becoming more and more researched over the last few years. This happens because the complexity in some video games, like Defense Of The Ancients 2 (DOTA 2), has been growing exponentially. Nowadays, this video game is one of the most difficult to play and one of the most complex, in terms of information to be processed by players in an entire match and also in terms of the number of decisions that can be made at a certain moment in a match. The difficulty present in DOTA 2 led to the increasing challenge and interest that is creating AIs for this game's agents.

Unfortunately, although there are some arguably effective AI implementations of DOTA 2 agents, there are no available open-source frameworks that can help a common user to develop this kind of implementation. This is why we created, in this thesis, a framework simple to use, with the objective of allowing AI development for DOTA 2 agents, that can ease the process of learning and testing the developed agents, using automatization through the Graphical User Interface (GUI) of this video game. This dissertation will be focused on all of the relevant aspects of this interface between the game and an external programming environment, using the Python programming language, namely the Architecture of this System, where all of the fundamental characteristics of this solution are defined, and the Implementation, where this system's process of development is detailed.

Conteúdo

Lista de Figuras

Lista de Tabelas

Siglas e Acrónimos

API	Application Programming Interface
CPU	Central Processing Unit
DLC	Downloadable Content
DLL	Dynamic-Link Library
DOTA 2	Defense Of The Ancients 2
FPS	First-Person Shooter
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
HTTP	HyperText Transfer Protocol
IA	Inteligência Artificial
ID	Identificador
ML	Machine Learning
MMR	Matchmaking Rating
MOBA	Multiplayer Online Battle Arena
RAM	Random-Access Memory
RL	Reinforcement Learning
SO	Sistema Operativo
UT	Unreal Tournament

Capítulo 1

Introdução

No início dos anos 50 começaram a ser propostos modelos e desenvolvidas IAs (Inteligências Artificiais) para jogos, com a motivação de, futuramente, virem a ser resolvidos problemas cada vez mais difíceis e/ou de maior complexidade [1]. Por problemas mais díficeis e complexos entenda-se com maior número de variáveis, estados e ações, podendo ser, estes últimos dois, discretos e/ou contínuos.

A IA está relacionada com o campo de ML (Machine Learning), no sentido em que uma máquina consiga aprender a comportar-se de uma forma "inteligente", aprendendo a adaptar-se a um determinado ambiente que seja mutável. Assim sendo, um programa de ML pode ser definido como um algoritmo ou conjunto de algoritmos que atuem sobre um problema e, com base em informação, dada (bases de dados) ou determinada (a partir de um modelo), escolhe uma solução, de um conjunto de várias soluções, de modo a resolver o problema [2]. ML pode ser aplicada a inúmeros problemas, nomeadamente classificação, predição de valores, reconhecimento de padrões, extração de conhecimento, ou até compressão de ficheiros [2]. Outra área de investigação relevante é a do entretenimento, em que se encontra o exemplo dos jogos de tabuleiro, nomeadamente o Xadrez [1], as Damas [3] e, mais recentemente, em 2017, o desenvolvimento de IA para o jogo Go [4].

Em 1950, Shannon propõe um modelo computacional para que, pela primeira vez na história, uma máquina consiga, sem ajuda humana, jogar uma partida completa de Xadrez, de uma forma habilidosa, sem ajuda de código forçado [1]. Shannon, ao constatar que

> "It is clear then that the problem is not that of designing a machine to play perfect chess [...] nor one which merely plays legal chess [...]. We would like to play a skilful game, perhaps comparable to that of a good human player." [1],

aponta para o, agora conhecido, termo de IA, ou seja, a capacidade de uma máquia tomar decisões consideradas "inteligentes" ou semelhantes às de uma pessoa considerada inteligente. O modelo proposto é composto por um conjunto de considerações a ter num programa para jogar Xadrez, funções de avaliação aproximadas para determinadas posições do tabuleiro, estratégias de Xadrez baseadas nas funções de avaliação e notas sobre como programar um computador para aplicar os elementos já citados. No mesmo artigo, Shannon tem a esperança de que o seu trabalho seja usado no futuro para aplicar a problemas

de maior dimensão e do mundo real, problemas "de maior significado" [1].

É com esta ideologia que, desde então, cresceu o interesse e a necessidade em aumentar a complexidade deste tipo de desafios. Por este motivo, têm vindo a desenvolver-se IAs mais avançadas para jogos mais complexos e dinâmicos, como são exemplos os videojogos StarCraft II [5, 6] e DOTA 2 (Defense Of The Ancients 2) [7, 8, 9, 10, 11].

O foco principal desta dissertação encontra-se no DOTA 2. Durante a realização deste trabalho foi desenvolvida uma *framework* que transforma a API (*Application Programming Interface*) oficial de *bots* do DOTA 2 [12] numa API escrita em Python, com o objetivo de facilitar o uso de ML e possibilitar implementações de agentes inteligentes.

1.1 Formulação do Problema

Com o aparecimento da API oficial de *scripting* de *bots* de DOTA 2 [12], no final do ano de 2016, surgiu um grande interesse, por parte da comunidade, de criar *scripts* para heróis de DOTA 2, que pudessem tomar ações com base em código construído pelos utilizadores desta API, sendo assim possível construir melhores *bots* para este jogo. Com a grande relevância que ML tem nos hoje em dia, não tardaria em ser associado este tipo de desenvolvimento de *scripts* para heróis de DOTA 2 com IA. Foi o que aconteceu meses mais tarde pela parte da OpenAI, que desenvolveu o primeiro agente de DOTA 2 [8, 9], capaz de derrotar os melhores jogadores do mundo no maior evento desportivo a nível de videojogos, o *The International* [13]. Passado um ano, em 2018, a OpenAI desenvolveu uma equipa de 5 heróis de DOTA 2, controlados por 5 agentes, para defrontar as melhores equipas do mundo, desta vez, na edição desse ano do *The International* [14].

Apesar de terem sido implementados agentes de DOTA 2 com enorme sucesso e eficácia, pela OpenAI, esta companhia sem fins lucrativos, que se propõe a ajudar o mundo através do desenvolvimento da área de ML [15], contribuindo com código aberto para múltiplos projetos de Inteligência Artificial, nunca chegou a publicar pelo menos um artigo científico sobre a implementação feita, quer em 2017, quer em 2018, de agentes de DOTA 2. Para além de não existir nenhum artigo científico sobre este tema, pela parte da OpenAI, esta companhia também nunca disponibilizou nenhuma ferramenta de *software* que possibilitasse aos fãs de DOTA 2, em particular, e de ML, em geral, o desenvolvimento de agentes para DOTA 2.

Para além da inexistência de uma *framework* disponibilizada pela OpenAI, também não existe, atualmente, qualquer tipo de APIs que possibilitem a criação de IA para este videojogo.

1.2 Solução Apresentada para o Problema

É devido às razões apresentadas na Secção 1.1 que se decidiu criar uma *framework* para desenvolvimento de agentes de DOTA 2. Esta ferramenta de *software* foi desenvolvida como trabalho de tese de Mestrado.

A *framework* tem como objetivo facilitar o desenvolvimento de agentes deste videojogo, através de um módulo, denominado `PythonDota2`, desenvolvido com recurso à linguagem de programação Python, fácil de utilizar, para que o seu utilizador não tenha de se preocupar com a criação de ambientes de partidas, obtenção do estado das mesmas e envio de comandos representativos de ações para os agentes a desenvolver.

A criação de ambientes de partidas e a sua automatização é efetuada através da GUI (*Graphical User Interface*) do DOTA 2, em que são criados *lobbies* locais, sendo necessário acesso à Internet, para assegurarem a criação de partidas. Todas as definições destes *lobbies* são configuradas através da automatização, para que o utilizador apenas tenha de fazer *plug and play*, ou seja, começar a utilizar o módulo sem uma grande quantidade de configurações.

A obtenção de estados das partidas é efetuada recorrendo a um processo, desenvolvido em Python, que trata de obter os ficheiros de *log* respetivos às partidas criadas, para onde são escritos os estados das partidas, através de um ficheiro LUA representativo do herói a ser controlado pelo agente a desenvolver, também desenvolvido no trabalho para esta tese, utilizando a API oficial de *scripting* de *bots* de DOTA 2 [12]. É efetuado *parsing* destes *logfiles*, de modo a obter informação sobre cada estado de cada partida do jogo. Neste processo são também atribuídos valores representativos do estado das partidas a variáveis presentes no módulo desenvolvido, para que seja possível ao utilizador aceder aos estados das partidas.

Finalmente, o envio de comandos é efetuado através de um procedimento que cria ficheiros LUA, contendo, cada ficheiro, um comando representativo de uma ação para o agente tomar. Estes ficheiros são lidos pelo *script* LUA desenvolvido, que trata de definir as ações respetivas para o agente, através da API oficial de *scripting* de *bots* de DOTA 2.

Assim, pretende-se que qualquer utilizador desta *framework*, possa, sem grande dificuldade, utilizar a mesma e começar a desenvolver IA para agentes deste videojogo, através de algoritmos de ML.

1.3 Estrutura da Dissertação

Além deste Capítulo, esta dissertação encontra-se dividida da seguinte forma:

No Capítulo 2 encontra-se uma descrição do videojogo DOTA 2 e da API de *scripting* de *bots* deste videojogo, bem como o estudo de duas implementações de agentes de DOTA 2. Ainda no mesmo Capítulo encontra-se uma análise de várias implementações de interfaces entre o DOTA 2 e alguns ambientes de programação. Finalmente é feito o estudo do estado da arte relacionado com a criação de interfaces para desenvolvimento de agentes noutros videojogos.

No Capítulo 3 é descrita a arquitetura e fluxo de informação do sistema desenvolvido. Ainda no mesmo Capítulo, encontra-se descrita a implementação da *framework*, onde é explicado o funcionamento do *script* LUA para um herói a controlar, tal como o processo de desenvolvimento do módulo `PythonDota2`, desenvolvido nesta tese.

No Capítulo 4 são apresentados e analisados os resultados de um conjunto de testes de

modo a se avaliar o sistema implementado. Os testes englobam o desempenho de leitura de ficheiros de *log* com a utilização de processos e de *threads*, a validação do sistema desenvolvido através da implementação de um agente desenvolvido em Python, um teste à consistência de dados obtidos através do módulo desenvolvido e ainda um teste de acesso a variáveis do estado de partidas através do módulo desenvolvido.

O Capítulo 5 apresenta uma reflexão sobre o trabalho desenvolvido, sobre as implementações estudadas no estado da arte desta dissertação e do trabalho futuro que pode ser realizado.

Capítulo 2

Estado da Arte

Neste Capítulo encontra-se uma descrição do videojogo DOTA 2, na Secção 2.1, que inclui detalhes sobre o mapa e unidades do jogo, bem como o funcionamento de partidas. Na Secção 2.2 encontra-se detalhada a API de *scripting* de *bots* de DOTA 2, nomeadamente os tipos de implementação possíveis e as funções que constituem a API.

Ainda neste Capítulo, na Secção 2.3 encontram-se detalhadas as implementações de um agente de DOTA 2 e de uma equipa composta por 5 agentes, bem como os resultados obtidos em cada uma das implementações.

Foi também efetuada uma análise de APIs já existentes para o videojogo DOTA 2, similares à que é proposta nesta dissertação, que se encontra na Secção 2.4. Estas APIs foram construídas tendo como base a API oficial de *bots* do DOTA 2 [12] bem como a API DOTA 2 *Workshop Tools* [16], que é utilizada para criar novos heróis, *skins* ou *sets* para os mesmos, novos itens ou até modos de jogo para o DOTA 2.

Por fim, na Secção 2.5, encontra-se o estudo do estado da arte efetuado sobre outras interfaces de ML aplicadas a videojogos. Foi efetuado um estudo sobre arquiteturas de APIs para videojogos (interfaces que facilitam o controlo de agentes (*bots*) em videojogos, programaticamente) mais antigos do que o DOTA 2, nomeadamente o jogo UT (Unreal Tournament) e o Quake II. No primeiro videojogo, foi desenvolvida uma interface, chamada *Gamebots* [17], que estabelece a ligação entre o motor de jogo do UT e uma infraestrutura multiagente que permite o controlo de vários agentes bem como acesso ao estado do ambiente que os rodeia (Subsecção 2.5.1) e, no segundo, foi desenvolvida uma interface para adicionar antecipação a um agente controlado por *software* no jogo Quake II [18] (Subsecção 2.5.2). Ambos os projetos têm como objetivo o progresso na investigação dentro do campo de ML, mais concretamente na área dos videojogos.

2.1 O Videojogo DOTA 2

DOTA 2 é um jogo de computador do género MOBA (Multiplayer Online Battle Arena), constituído por duas equipas, chamadas *Radiant* e *Dire*. Cada uma das equipas possui cinco elementos principais, chamados "heróis", que podem ser controlados por humanos

ou por IAs. Cada uma das duas equipas tem como objetivo tentar destruir o *Ancient* adversário, de modo a vencer a partida. Cada equipa possui um *Ancient*, que é o edifício mais importante do jogo, encontrando-se localizado na base respetiva a cada equipa.

2.1.1 O Mapa de DOTA 2

O mapa de DOTA 2 tem uma estrutura quadrangular e encontra-se dividido ao meio, por um rio, que corre de Noroeste para Sudeste, em duas partes triangulares onde, em cada uma delas, existe uma base. As equipas *Radiant* e *Dire* começam uma partida de DOTA 2 nas suas bases, a Sudoeste e a Nordeste do rio, respetivamente. Para além das duas bases e do rio, o mapa do jogo é constituído por outras doze partes:

- A *Top Lane* (Caminho de Cima), dividida em duas partes:
  - Parte dos *Radiant*, localizada de Sudoeste a Noroeste. Esta localização encontra-se abreviada pela sigla "T", de cor azul clara, na Figura 2.1;
  - Parte dos *Dire*, localizada de Noroeste a Nordeste. Esta localização encontra-se abreviada pela sigla "T", de cor vermelha, na Figura 2.1.
- A *Bottom Lane* (Caminho de Baixo), dividida em duas partes:
  - Parte dos *Radiant*, localizada de Sudoeste a Sudeste. Esta localização encontra-se abreviada pela sigla "B", de cor azul clara, na Figura 2.1;
  - Parte dos *Dire*, localizada de Sudeste a Nordeste. Esta localização encontra-se abreviada pela sigla "B", de cor vermelha, na Figura 2.1.
- A *Middle Lane* (Caminho do Meio), dividida em duas partes:
  - Parte dos *Radiant*, localizada de Sudoeste até ao centro do mapa. Esta localização encontra-se abreviada pela sigla "M", de cor azul clara, na Figura 2.1;
  - Parte dos *Dire*, localizada desde o centro do mapa até Nordeste. Esta localização encontra-se abreviada pela sigla "M", de cor vermelha, na Figura 2.1.
- A *Jungle* (Floresta), dividida em duas partes:
  - Parte dos *Radiant*, que se encontra nas zonas entre o rio e os caminhos de cima e de baixo da mesma equipa. Esta localização encontra-se abreviada pelas siglas "J", de cor azul clara, na Figura 2.1;
  - Parte dos *Dire*, que se encontra nas zonas entre o rio e os caminhos de cima e de baixo da mesma equipa. Esta localização encontra-se abreviada pelas siglas "J", de cor vermelha, na Figura 2.1.
- As *Shops* (Lojas), que são de três tipos:

Figura 2.1: Mapa de uma partida de DOTA 2, simplificado [19]. No mapa estão representadas variadas siglas que abreviam localizações do mapa. Estas localizações encontram-se devidamente descritas na Subsecção 2.1.1. As siglas de cor azul clara correspondem a localizações na parte do mapa da equipa *Radiant*, as siglas de cor vermelha correspondem a localizações na parte do mapa da equipa *Dire* e as siglas de cor azul escura representam localizações neutras.

– As *Base Shops* (Lojas das Bases), que são duas, localizadas nas respetivas bases das equipas. Estas localizações encontram-se abreviadas pelas siglas "BS", de cores azul clara e vermelha, na Figura 2.1;

– As *Side Lane Shops* (Lojas Laterais aos Caminhos), que são duas. Encontram-se, tal como o nome sugere, lateralmente à *top lane* dos *Radiant*, a Noroeste, e lateralmente à *Bottom Lane* dos *Dire*, a Sudeste. Estas localizações encontram-se abreviadas pelas siglas "SLS", de cores azul clara e vermelha, na Figura 2.1;

– As *Secret Shops* (Lojas Secretas), que são duas. Uma encontra-se na *Jungle* dos *Radiant*, entre a *Top Lane* e *Middle Lane*. A outra encontra-se na *Jungle*

dos *Dire*, entre a *Middle Lane* e *Bottom Lane*. Estas localizações encontram-se abreviadas pelas siglas "SS", de cores azul clara e vermelha, na Figura 2.1.

- O *Roshan Pit* (Abismo de Roshan), que se encontra no rio, mais concretamente entre a *Top Lane* e a *Middle Lane*; Estas localização encontra-se abreviada pela siglas "RP", de cor azul escura, na Figura 2.1;

- *Rune Spawning Locations* (Locais de Aparecimento de Runas), que são de dois tipos:

  - *Power-up Rune Locations* (Locais de Runas de Aumento de Potência). Estas runas encontram-se em duas localizações no rio, estando uma diretamente por baixo do *Roshan Pit*, e outra entre as *Jungles* das duas equipas, entre a *Middle Lane* e a *Bottom Lane*. Estas localizações encontram-se abreviadas pelas siglas "PRL", de cor azul escura, na Figura 2.1;

    Existem seis tipos de *Power-up Runes*, nomeadamente:

    * *Arcane* (Arcana), que reduz os *cooldowns* de itens e habilidades especiais do herói que ativou a runa, durante cinquenta segundos. *Cooldown* é o termo usado para indicar que o item ou habilidade especial, quando usado, tem um tempo de espera definido até ser usado novamente;
    * *Double Damage* (Dano Duplo), que aumenta o dano base do herói que ativou a runa, durante quarenta e cinco segundos;
    * *Haste* (Celeridade), que aumenta para o máximo a rapidez de movimento do herói que ativou a runa, durante vinte e dois segundos;
    * *Illusion* (Ilusão), que cria duas ilusões do herói que ativou a runa, durante setenta e cinco segundos;
    * *Invisibility* (Invisibilidade), que torna o herói que ativou a runa invisível, durante quarenta e cinco segundos;
    * *Regeneration* (Regeneração), que recupera totalmente 6% dos pontos de vida máximos, bem como 6% da *mana* máxima do herói que ativou a runa. Esta runa tem uma duração máxima de trinta segundos. É interrompido o efeito se o herói sofrer qualquer tipo de dano ou quando estiver totalmente recuperado;

  - *Bounty Rune Locations* (Locais de Runas de Recompensa). Estas runas encontram-se em quatro localizações. Duas delas encontram-se perto de cada uma das *Secret Shops*. As outras duas encontram-se em cada uma das *Jungles*, estando uma delas localizada na *Jungle* dos *Radiant*, entre a *Middle Lane* e a *Bottom Lane*, e a outra localizada na *Jungle* dos *Dire*, entre a *Top Lane* e a *Middle Lane*. Estas localizações encontram-se abreviadas pela sigla "BRL", de cores azul clara e vermelha, na Figura 2.1.

    Este tipo de runa recompensa cada herói da equipa do herói que a ativou com quarenta unidades de ouro mais duas por cada minuto decorrido durante a partida. Por exemplo, aos zero segundos de jogo, ativando uma *Bounty Rune*,

cada herói da equipa recebe quarenta unidades de ouro. Se a runa tivesse sido ativada apenas aos cento e vinte segundos (dois minutos) de jogo, cada herói receberia quarenta mais o dobro de duas unidades de ouro, perfazendo um total de quarenta e quatro unidades de ouro recebidas por herói.

Outros dois aspetos importantes relacionados, direta e indiretamente, com o mapa de DOTA 2 são a visão [20] que uma equipa tem do mapa e os sons da partida ouvidos pelos heróis de uma equipa.

A visão de um herói, no mapa de DOTA 2, depende da sua localização, bem como dos obstáculos que se encontram no seu raio de visão. A esta limitação dá-se o nome *Fog of War* (Nevoeiro de Batalha). A maior parte do Nevoeiro de Batalha é criado pelas árvores que compõem o mapa, que impedem a visão do que está por detrás delas. A visão de um herói também é limitada pela elevação de um herói, ou seja, se este se encontrar por baixo de, por exemplo, uma colina, não conseguirá ver o que está a acontecer ao nível da mesma. No entanto, quanto maior a elevação de um herói, no mapa, maior será a sua visão para níveis inferiores ao terreno em que se encontra. De modo a aumentar a visão de jogo, um herói tem a possibilidade de comprar um item, chamado *Observer Ward* (Sentinela), que aumenta a visão do mapa para uma equipa, numa área circular. A visão do mapa é maior de dia do que é de noite. É possível destruir *Observer Wards* adversárias com recurso a *Sentry Wards* (Sentinelas de Vigia). Todas as unidades de uma equipa fornecem visão sobre o mapa.

Quanto ao som, um jogador humano pode aperceber-se de que algo aconteceu perto do herói controlado por si, através do som da partida, mesmo que não tenha visão sobre esse acontecimento. Isto não se aplica a um herói controlado por *software*.

2.1.2 Unidades e Funcionamento de Partidas de DOTA 2

Cada equipa é constituída por elementos chamados unidades. Um elemento é uma unidade se possuir, pelo menos, HP (*Health Points*), ou, em português, pontos de vida, bem como armadura. Existem, portanto, 3 tipos de unidades:

- *Heroes* ou "Heróis", cinco em cada equipa. Estas unidades podem ser *melee*, ou seja, atacam corpo a corpo, ou *ranged*, ou seja, atacam à distância;
- Edifícios, nomeadamente:
  - *Ancients*, um em cada base. São os edifícios mais importantes. Se uma equipa destruir o *Ancient* adversário, vence a partida;
  - Torres, onze de cada equipa, que defendem uma determinada área à sua volta;
  - *Barracks* ou "Quartéis", dois de cada equipa;
  - *Fountains* ou "Fontes", uma em cada base. Os heróis começam a partida neste edifício, nas respetivas bases. Heróis e unidades aliadas (e.g., unidades controladas por heróis, como é exemplo do estafeta), quando presentes no raio deste edifício, regeneram os seus HP e *mana*;

  - *Shrines* ou "Santuários", dois de cada equipa. Heróis e unidades aliadas, quando presentes no raio deste edifício, obtêm o mesmo benefício da *Fountain*;
  - Edifícios anexos presentes nas bases das equipas. Heróis e unidades aliadas, quando presentes no raio destes edifícios, obtêm o mesmo benefício da *Fountain*.

- *Creeps*, nomeadamente:
  - *Creeps* de *lane*, unidades de combate auxiliares, que possuem quatro vertentes:
    * *Melee creeps*;
    * *Ranged creeps*;
    * *Siege creeps*, que são máquinas que catapultam pedras;
    * *Megacreeps*, que são evoluções dos *creeps* normais de *lane*. Este tipo de *creeps* substitui o tipo normal de *creeps* de uma equipa, se esta destruir os quartéis da base adversária.
  - *Creeps* neutros, presentes em sete campos na *jungle* dos *Radiant* e noutros sete campos na *jungle* dos *Dire*;
  - *Ancient Creeps*, presentes em dois campos na *jungle* dos *Radiant* e noutros dois campos na *jungle* dos *Dire*.
  - *Roshan*, que é um *creep* neutro especial, localizado no *Roshan pit*. Quando morto, *Roshan* liberta um item (*Aegis of the Immortal*) que possibilita, a um herói, renascer, passados cinco segundos, na mesma localização da sua morte. Este item só tem uma duração de cinco minutos depois de ter sido apanhado [21]. *Roshan* é a primeira unidade, juntamente com os heróis, a ser criada quando se dá início a uma partida.
  - *Couriers* ou "estafetas", voadores ou não, que têm a função de entregar itens aos heróis.

Os atributos base mais importantes de um herói são: HP, *mana* (permite usar habilidades especiais), armadura, resistência a magia, rapidez de movimento e rapidez de ataque. Um herói é categorizado como herói de Força, Agilidade ou Inteligência, por ter mais pontos de uma destas categorias em relação às outras duas [22]. Pontos de Força aumentam os HP máximos de um herói, bem como a percentagem de regeneração de HP. Pontos de Agilidade aumentam a armadura de um herói, bem como a sua rapidez de ataque. Pontos de Inteligência aumentam a *mana* máxima de um herói, bem como a percentagem de regeneração de *mana*. Os heróis têm habilidades especiais, que podem ser passivas ou ativas: as habilidades passivas têm efeitos que estão sempre ativos, portanto não necessitam de ser ativadas [23]; as habilidades ativas, como o próprio nome sugere, têm de ser ativadas podendo ser necessário escolher um alvo, que pode ser uma área ou uma unidade.

Uma partida de DOTA 2 começa pela fase de escolha de heróis, em que se escolhem, cinco heróis para cada equipa. Segue-se a criação dos heróis no mapa e, consequentemente, a fase do tempo de atraso da partida, que tem uma duração de um minuto e trinta segundos.

Durante esta fase, os heróis preparam a sua estratégia inicial e podem circular no mapa, mesmo ainda sem a partida ter começado. Nesta fase, se heróis adversários se confrontarem, podem atacar-se e quaisquer mortes resultantes são contabilizadas, bem como o ouro e a experiência ganhos. Depois desta fase começa a partida. Uma vez iniciada a partida, os *creeps* de *lane* "nascem" duas vezes por minuto (aos zero e trinta segundos), em cada uma das três *lanes*. Estas unidades percorrem a *lane* respetiva, onde encontrarão *creeps* de *lane*, torres e outros edifícios e, provavelmente, heróis adversários. Cada equipa tenta destruir as torres adversárias, matar *creeps* e heróis da outra equipa de modo a ganhar ouro e experiência durante uma partida. O ouro é utilizado para comprar itens que se adaptem a cada herói, de maneira a torná-lo mais potente ou mais útil para a sua equipa. A experiência é utilizada para aumentar o nível do herói, sendo 25 o nível máximo. Quando um herói sobe de nível tem a possibilidade de aumentar o nível das suas habilidades especiais. Nos níveis 10, 15, 20 e 25, um herói pode selecionar um de dois "talentos" únicos (nenhum outro herói possui a mesma escolha de "talentos"), que lhe são apresentados [24]. Estes "talentos" têm a finalidade de melhorar certos aspetos do herói. A partida termina quando uma equipa destruir o *Ancient* adversário, sendo, assim, a equipa vencedora.

2.2 *DOTA 2 Bot API*

Nesta Secção encontram-se descritos os tipos de implementação possíveis em *bots* de DOTA 2, com o uso da API de *scripting* de *bots* deste jogo, bem como os tipos de funções desta API.

2.2.1 Tipos de Implementação de *Bots* Fornecidos pela *DOTA 2 Bot API*

A API de *bots* de DOTA 2 [12] fornece, ao utilizador, uma interface entre o jogo e a linguagem de *scripting* LUA.

Existem sete tipos de implementação de lógica em *scripts* LUA, para DOTA 2:

- *Complete Takeover* (Controlo Total). Usando este tipo de implementação, o utilizador da API tem o controlo total de um herói a partir de um *script* denominado `bot_<nome_do_heroi>.lua`. Neste *script* é apenas substituída a função `Think()` do herói, por uma implementação nova, substituindo, assim, o controlo predefinido do herói (IA padrão), pelo novo controlo implementado.

- *Mode Override* (Controlo de Modos). Neste tipo de implementação existem várias funções que podem ser controladas. Por exemplo, se o utilizador desejar alterar o modo de ataque para um herói, terá de criar um ficheiro chamado `mode_attack_<nome_do_heroi>.lua`, onde poderá alterar quatro funções, nomeadamente:

  - `GetDesire()`, que inquire, ao motor de jogo do DOTA 2, sobre a importância de o modo *attacking* ser o modo ativo em determinado momento;

- `OnStart()`, que é utilizada quando o modo *attacking* é o modo ativo;
- `OnEnd()`, que é utilizada quando o modo *attacking* deixa de ser o modo ativo;
- `Think()`, que é utilizada para definir ações para o herói executar, quando *attacking* é o modo ativo.

Para além do modo *attacking* é possível assumir o controlo de outros vinte modos, nomeadamente:

- *laning*, utilizado para controlar o herói durante a *laning phase* que é a fase inicial do jogo, em que cada herói é atribuído a uma *lane*;
- *roam*, para controlar o herói, de modo a que este circule pelo mapa;
- *retreat*, para controlar o herói a recuar;
- *secret_shop*, para controlar o deslocamento do herói até uma das duas *secret shops*;
- *side_shop*, para controlar o deslocamento do herói até uma das duas *side shops*;
- *rune* para controlar o deslocamento do herói até uma das seis localizações de aparecimento de runas e, eventualmente, apanhá-las;
- *push_tower_top*, para controlar o herói, de modo a que ataque a torre atacável da *top lane*;
- *push_tower_mid*, para controlar o herói, de modo a que ataque a torre atacável da *middle lane*;
- *push_tower_bot*, para controlar o herói, de modo a que ataque a torre atacável da *bottom lane*;
- *defend_tower_top*, para controlar o herói, de modo a que defenda a torre a ser atacada na sua *top lane*;
- *defend_tower_mid*, para controlar o herói, de modo a que defenda a torre a ser atacada na sua *middle lane*;
- *defend_tower_bottom*, para controlar o herói, de modo a que defenda a torre a ser atacada na sua *bottom lane*;
- *assemble*, para controlar o herói, de modo a que se junte com os heróis aliados;
- *team_roam*, para controlar o herói, de modo a que este circule pelo mapa com o resto da equipa.
- *farm*, para controlar o herói a executar *farming*. *Farming* é a mecânica de jogo que consiste em matar *creeps* com o último golpe, de modo a obter experiência e ouro;
- *defend_ally*, para controlar o herói, de modo a que defenda um herói aliado;
- *evasive_maneuvers*, para controlar o herói, de modo a que execute manobras evasivas, durante uma luta entre equipas;

- *roshan*, para controlar o deslocamento do herói até ao *Roshan Pit* e, eventualmente, atacar Roshan;
- *item*, para controlar o herói, de modo a que utilize determinados itens;
- *ward*, para controlar o herói, de modo a que coloque *wards* em determinadas localizações, para a equipa obter maior visão sobre o mapa.

- *Ability and Item Usage* (Uso de Habilidades e Itens). É possível apenas alterar as tomadas de decisão dos heróis, bem como o uso de itens e habilidades especiais e tomadas de decisão do estafeta principal da equipa. Existem cinco funções neste tipo de implementação que podem ser reimplementadas, nomeadamente:
  - `ItemUsageThink()`, que é utilizada para controlar o uso de itens, pelo herói;
  - `AbilityUsageThink()`, que é utilizada para controlar o uso de habilidades especiais, pelo herói;
  - `CourierUsageThink()`, que é utilizada para controlar as ações do estafeta principal da equipa;
  - `BuybackUsageThink()`, que é utilizada para controlar a ação de *buyback*, do herói. *Buyback* é o nome de uma ação que um herói pode tomar, que consiste em gastar ouro para renascer imediatamente, na sua base, sem que tenha de esperar pelo *cooldown* enquanto está morto. Só é possível executar esta ação a cada quatrocentos e oitenta segundos e o seu custo é determinado pela equação 2.1 [25]:
    $$Custo = 100 + Networth/13 \tag{2.1}$$
    em que *Networth* é o valor líquido de ouro acumulado pelo herói em determinado momento;
  - `AbilityLevelUpThink()`, que é utilizada para controlar o aumento de determinada habilidade especial, do herói. Para além deste custo, o próximo renascimento do herói, na sua base, demorará mais vinte e cinco segundos do que o que seria normal.

  As funções anteriormente descritas podem ser implementadas num ficheiro chamado `ability_item_usage_<nome_do_heroi>.lua`.

- *Minion Control* (Controlo de *Minions*). É possível apenas assumir controlo de *minions* aliados, nomeadamente ilusões de heróis, *creeps* convocados por heróis e *creeps* dominados por heróis, através do uso do item *Helm of the Dominator*. Para implementar este controlo é efetuado um *override* da função `MinionThink()` no *script* `bot_<nome_do_heroi>.lua`, de maneira a controlar as unidades, já referidas, de um herói;

- *Item Purchasing* (Compra de Itens). É possível apenas assumir controlo da compra de itens de um herói. Para implementar a tomada de decisão que envolve a

compra de itens, é possível implementar a função `ItemPurchaseThink()` no *script* `item_purchase_<nome_do_heroi>.lua`.

- *Team Level Desires* (Desejos ao Nível da Equipa). É possível apenas alterar os desejos a nível da equipa. Existem seis funções neste tipo de implementação que podem ser reimplementadas, nomeadamente:

  - `TeamThink()`, utilizada para implementar a lógica principal para a equipa inteira;
  - `UpdatePushLaneDesires()`, utilizada para atualizar os desejos de pressionar cada uma das três *lanes*;
  - `UpdateDefendLaneDesires()`, utilizada para atualizar os desejos de defender cada uma das três *lanes*;
  - `UpdateFarmLaneDesires()`, utilizada para atualizar os desejos de executar *farming* em cada uma das três *lanes*;
  - `UpdateRoamDesire()`, utilizada para atualizar o desejo de um herói circular pelo mapa e emboscar um herói adversário;
  - `UpdateRoshanDesire()`, utilizada para atualizar o desejo de a equipa matar Roshan.

  Para implementar a lógica dos dejesos da equipa, é possível implementar estas funções no *script* denominado `team_desires.lua`.

- *Hero Selection* (Seleção de Heróis). É possível apenas alterar a escolha de heróis e as suas funções, em termos de *lanes*, utilizando três funções, nomeadamente:

  - `Think()`, utilizada para selecionar os heróis para os *bots*;
  - `UpdateLaneAssignments()`, utilizada para atribuir *lanes* a heróis;
  - `GetBotNames()`, utilizada para obter os nomes dos jogadores.

  Para implementar a lógica da seleção de heróis, é possível implementar estas funções no *script* denominado `hero_selection.lua`.

A nossa implementação assume o tipo de implementação *Complete Takeover*, pois é utilizado apenas o *script* `bot_<nome_do_heroi>.lua` com a função `Think()`, para implementar a interação com um módulo desenvolvido em Python, através do qual o herói é controlado. Para além deste *script*, foi também utilizado o ficheiro `hero_selection.lua` para atribuir os heróis às equipas, visto que, na nossa implementação, foi necessária a criação de *lobbies* locais para a criação de ambientes de jogo. Os *lobbies* lêm a informação do ficheiro `hero_selection.lua` e criam a partida com os heróis respetivos em cada equipa.

2.2.2 Funções da API

Nesta API, existem três tipos de funções que podem ser usadas:

- *Global Functions* (Funções Globais), que têm, sobretudo, utilização para obter informações globais do estado do jogo;
- *Unit Scoped Functions* (Funções do Âmbito das Unidades), que são utilizadas para obter informação do estado de unidades aliadas ou adversárias e para unidades aliadas executarem ações;
- *Ability and Item Scoped Functions* (Funções no Âmbito das Habilidades Especiais e dos Itens), que são utilizadas para obter informação sobre o estado dos itens e de habilidades especiais do herói controlado, tal como de heróis aliados.

As únicas funções globais utilizadas, na nossa implementação, para a construição do estado do jogo foram:

- { *hUnit, ...* } `GetUnitList( nUnitType )`. Esta função retorna *handles* para unidades. O argumento `nUnitType` indica de que tipo de unidades se pretendem os *handles*; `nUnitType` pode ser, por exemplo, "UNIT_LIST_ALLIED_HEROES", de modo a obter *handles* para os heróis aliados;
- *hUnit* `GetBot()`. Esta função retorna um *handle* para o herói controlado pelo *script* LUA;
- *float* `DotaTime()`. Esta função retorna o tempo de jogo da partida de DOTA 2, sendo o tipo de dados *float*, em segundos;
- *float* `GetUnitToUnitDistance( hUnit1, hUnit2 )`. Esta função retorna a distância (numa medida inerente ao mapa do jogo) entre duas unidades, sendo o tipo de dados *float*. Os argumentos da função são *handles* para as unidades entre as quais se pretende medir a distância;
- *float* `GetTimeOfDay()`. Esta função retorna um valor entre 0 e 1, que representa a altura do dia. Se o valor for menor do que 0.25 ou maior do que 0.75, é de noite; se o valor estiver entre 0.25 e 0.75, é de dia;
- *int* `GetGameState()`. Esta função retorna o estado da partida, sendo o tipo de dados *int*, podendo ser retornados valores de 0 a 10, consoante o estado em que a partida se encontra.

Foram utilizadas várias funções do âmbito das unidades, de modo a obter variadas informações sobre as mesmas, nomeadamente HP, *mana* e armadura. Este tipo de funções foi utilizado com base nos *handles* obtidos através das funções globais { *hUnit, ...* } `GetUnitList( nUnitType )` e *hUnit* `GetBot()`. Algumas funções do âmbito das unidades, que são utilizadas para definir ações para o *bot*, funcionam com base numa *double*

ended queue (fila duplamente terminada) (Figura 2.2). Esta estrutura de dados abstrata permite adicionar elementos à cabeça ou à cauda da fila, sendo os elementos ações, que são removidas quando começam a ser executadas ou quando é enviado um comando para limpar a fila, podendo ser abortada a ação a ser executada, em determinado momento, fazendo parar o herói, ou deixando que a ação continue a ser executada até terminar. Tomando o exemplo da ação *MoveToLocation*, que permite ao herói deslocar-se até uma dada localização, é possível usar três funções para esta ação, nomeadamente:

- `Action_MoveToLocation( vLocation )`. Esta função não interage com a *double ended queue* e executa apenas a ação de se movimentar até à localização `vLocation`, que é dada por um Vector de coordenadas (x, y, z). Quaisquer ações que estivessem na *double ended queue*, de movimentação do *bot* ou não, são eliminadas e é executada esta função.

- `ActionPush_MoveToLocation( vLocation )`. Esta função interage com a *double ended queue*, adicionando à cabeça da fila a ação *MoveToLocation*, tendo como argumento a `vLocation`. Quaisquer ações que estivessem na *double ended queue*, de movimentação do *bot* ou não, permanecem na fila, mas a nova ação será executada primeiro. Quaisquer novas ações que sejam *pushed* (empurradas) para o início da fila serão sempre as ações que serão executadas em primeiro lugar, sendo a última que é *pushed* a primeira a ser executada.

- `ActionQueue_MoveToLocation( vLocation )`. Esta função interage com a *double ended queue*, adicionando à cauda da fila a ação *MoveToLocation*, tendo como argumento a `vLocation`. Quaisquer ações que estivessem na *double ended queue*, de movimentação do *bot* ou não, permanecem na fila, sendo a nova ação adicionada a última a ser executada, por se encontrar no fim da fila. Quaisquer novas ações que sejam *queued* (postas no fim da fila) serão sempre as ações que serão executadas em último lugar, sendo a última que é *queued* a última a ser executada.

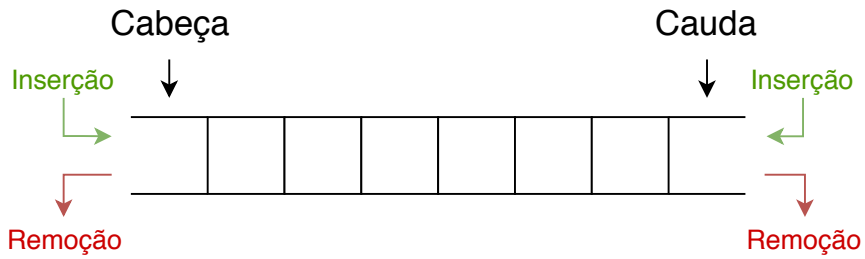


Figura 2.2: Funcionamento de uma fila duplamente terminada.

É possível eliminar todas as ações na fila, utilizando a função `Action_ClearActions( bStop )`, em que o argumento `bStop` é do tipo de dados *boolean* e, se for `True`, o herói para o seu movimento (se estiver a movimentar-se) quando é executada a função. Se `bStop` for `False`, apenas são eliminadas as ações da fila.

Existe, também, outro tipo de funções do âmbito das unidades que são imediatas, ou seja, não necessitam de interagir com a *double ended queue* em que a ação é executada imediatamente, mesmo que esteja a ser executada uma ação proveniente da fila de ações. Um exemplo deste tipo de funções imediatas é a função `int ActionImmediate_PurchaseItem ( sItemName )`, que define uma ação para o agente efetuar a compra de um item com o valor do argumento `sItemName`, que é do tipo de dados *string*. Esta função retorna um valor inteiro, representante do resultado da compra do item.

Existem dois conjuntos de funções que não tem qualquer tipo de utilidade no modo de implementação *Complete Takeover*. Um dos conjuntos é composto pelas Funções Globais:

- `float GetPushLaneDesire( nLane );`
- `float GetDefendLaneDesire( nLane );`
- `float GetFarmLaneDesire( nLane );`
- `float GetRoamDesire();`
- `hUnit GetRoamTarget();`
- `float GetRoshanDesire();`

O outro conjunto é composto pelas Funções do Âmbito das Unidades:

- `SetTarget( hUnit );`
- `hUnit GetTarget();`
- `SetNextItemPurchaseValue( nGold );`
- `int GetNextItemPurchaseValue();`
- `int GetActiveMode();`
- `float GetActiveModeDesire().`

Nenhuma das funções listadas anteriormente tem usabilidade em *Complete Takeover* porque, para as usar, assume-se que é feito *Mode Override*, ou uma variante do mesmo, em que seja possível, por exemplo, só alterar os desejos a nível de uma equipa (género de implementação *Team Level Desires*).

2.3 Agentes de DOTA 2

A OpenAI é uma companhia, fundada em 2015, sem fins lucrativos, que tem como atividade a investigação na área de IA. Esta companhia tem como objetivo "proporcionar avanços na inteligência digital, de uma forma que seja mais provável beneficiar a humanidade, como um todo" [26].

Esta companhia afirma que nunca privará o seu público de informação, para benefício próprio [15]. No entanto, esta afirmação não é inteiramente verdadeira, pois não existe nenhuma ferramenta disponibilizada pela OpenAI, nem mesmo através da ferramenta de *software* Gym [27], que permita o desenvolvimento de agentes de DOTA 2. Gym é um produto *open-source* (de código aberto), bastante recente, utilizado na área de ML e é desenvolvido e atualizado pela OpenAI. Esta ferramenta de *software* proporciona o desenvolvimento de algoritmos de RL (*Reinforcement Learning*), sem que o utilizador tenha de despender tempo a criar ambientes para determinados problemas, pois estes já se encontram presentes no Gym.

Tal como não existe nenhuma ferramenta de *software* que permita a interação, programaticamente, com o DOTA 2, fornecida pela OpenAI, também não existe qualquer documentação oficial (artigos científicos ou publicações em revistas, por exemplo) sobre a construção da interface entre o DOTA 2 e o desenvolvimento de algoritmos para agentes de DOTA 2, que já foi efetuado pela OpenAI em 2017 e 2018. As informações existentes (muito simplificadas para os leitores menos instruídos sobre ML) sobre a implementação de agentes deste jogo encontram-se apenas no blogue oficial desta companhia [8, 9, 10], onde é apenas referido que, para ambas as implementações de agentes de DOTA 2 [8, 10], realizadas pela OpenAI, a estratégia de desenvolvimento utilizada, para os algoritmos dos agentes, foi *self-play*.

Self-play é uma estratégia de aprendizagem utilizada em *Reinforcement Learning* (RL) que não é, portanto, supervisionada, nem utiliza dados provenientes de experiências humanas [4]. Esta técnica é aplicada de modo a que o agente comece por explorar o seu ambiente de uma forma aleatória, aprendendo por si próprio a tomar as melhores ações para determinados estados, com o passar do tempo de aprendizagem. Esta metodologia de ML não é recente; foi aplicada em 1995 ao jogo de Gamão (*backgammon*), para treinar uma rede neuronal, funcionando esta como função de avaliação para o mesmo jogo, jogando contra si própria e aprendendo através dos seus resultados [28]. Mais recentemente, esta metodologia foi utilizada no programa *AlphaGo Zero* [4] de modo a treinar um agente para o jogo Go, bem como para o desenvolvimento de agentes de DOTA 2 [8, 9, 29].

Nas Subsecções que se seguem, encontra-se uma breve descrição da aplicação de *self-play* aos algoritmos de RL, utilizados para o desenvolvimento de agentes de DOTA 2, feito pela OpenAI, bem como a implementação de um único herói (Subsecção 2.3.1) e de uma equipa inteira (Subsecção 2.3.2).

2.3.1 OpenAI 1v1

A aplicação de *self-play* em RL para agentes de DOTA 2 foi desenvolvida pela OpenAI, inicialmente, em 2017, para apenas um agente jogar contra um herói adversário, igual ao herói controlado pelo agente, denominado *Shadow Fiend*, sendo este herói controlado por um humano.

Na implementação deste agente foram focadas algumas das mecânicas de jogo mais importantes para um bom jogador de DOTA 2 e aplicadas na aprendizagem do agente. As mecânicas são as seguintes:

- *Last Hitting*, que consiste no ato de um herói dar o último golpe numa unidade adversária, de modo a receber experiência e ouro;
- *Creep Aggro Management*, que consiste no ato de um herói atacar um herói adversário de modo a atrair os *creeps* adversários;
- *Creep Blocking*, que consiste no ato de um herói abrandar a passagem dos *creeps* pela *lane*, de modo a obrigar os *creeps* adversários a avançar mais do que seria suposto;
- *Zoning*, que consiste no ato de um herói se posicionar de uma forma ameaçadora na *lane*, de modo a que o herói adversário tenha receio de estar perto dele e, consequentemente, receio de efetuar *last hits*;
- *Dodging*, que consiste no ato de um herói se esquivar a habilidades especiais efetuadas por heróis adversários em áreas do mapa;
- *Faking*, que consiste no ato de um herói fingir que irá executar um ataque ou uma habilidade especial. Esta mecânica é efetuada quando se para a ação do herói;
- *Chasing*, que consiste no ato de um herói perseguir um herói adversário pelo mapa, de forma a matá-lo;
- Adaptação a novas situações, que não tenham sido aprendidas pelo agente, durante a fase de aprendizagem, e que, portanto, lhe sejam estranhas.

O agente obtinha o estado do jogo e executava ações através da API de *bots* de DOTA 2. Para além disto, recebia "incentivos" por vitória e por desempenho em cada partida.

A implementação começou a 1 de março de 2017, com um cenário bastante simples. O agente foi progredindo e testado contra jogadores de baixo nível, inicialmente, até jogadores de topo, a nível mundial. Nos últimos três dias até dia 11 de agosto de 2017, o agente apresentou melhorias de dia para dia, derrotando adversários cada vez melhores, tendo no último dia sido vitorioso em 60% das vezes, contra o *bot* do dia anterior.

Os melhores jogadores que testaram o *bot* criado aperceberam-se que podiam explorar algumas falhas do mesmo e, por isso, tentaram inovar na sua abordagem ao jogo. Uma destas abordagens era comprar itens pouco comuns para uma fase inicial do jogo, que forneciam maior velocidade ao herói controlado pelo humano e afetavam a velocidade do

bot, quando atacado, diminuindo a sua rapidez. Assim os jogadores podiam perseguir o *bot* e matá-lo com facilidade, no início da partida. Outra abordagem utilizada, pelos jogadores, foi desenvolver uma habilidade especial do herói *Shadow Fiend* que dá mais dano ao adversário, do que ataques normais, na fase inicial da partida. Assim, o *bot*, como não esperava que o adversário utilizasse a habilidade especial logo no início do jogo, não conseguia reagir prontamente e morria. O facto de um herói morrer nesta fase do jogo implica imediatamente uma vantagem muito maior para o adversário, que terá maior probabilidade de vencer a partida.

Tendo desenvolvido um agente para DOTA 2, a OpenAI, propôs como trabalho futuro desenvolver uma equipa de cinco elementos, que pudesse vir a competir contra equipas profissionais no *The International* de 2018 [13].

2.3.2 OpenAI Five

A OpenAI Five, composto por cinco redes neuronais, treina um total de 180 dias de jogo em apenas um dia, utilizando *self-play*. Este desempenho é apenas possível devido à sua execução em 256 GPUs (*Graphics Prcessing Units*) e 128000 núcleos de CPU (*Central Prcessing Unit*) [10]. Tal como no OpenAI 1v1 [8, 9], não existe qualquer informação de experiências humanas, devido ao uso de *self-play*. Utilizando apenas esta estratégia de aprendizagem em RL, os agentes, como equipa, aprenderam estratégias de jogo de grande nível.

O problema identificado pela OpenAI para a construção de uma equipa composta por 5 agentes controlados por *software* divide-se em 4 componentes:

- Grande duração de partidas, que, em média, duram 45 minutos. Em cada segundo existem 30 *ticks*, portanto, no total, uma partida pode ter mais do que 80000 *ticks*. Neste sistema, as ações e estado do jogo são renovados a cada quatro *ticks*, reduzindo para um quarto o número de *ticks* fundamentais;

- Estado parcialmente observado, significando que apenas se conhece o estado do jogo visível, fornecido pelas unidades aliadas de uma equipa. O estado não observável não é, portanto, visível e tem o nome de *Fog of War* ou Nevoeiro de Batalha;

- Espaço de ações de grande dimensão, aproximandamente 1000 ações por *tick*. Comparativamente, o jogo de Xadrez é constituído por, em média, 35 ações e o jogo de Go por 250 ações.

- Espaço de estados de grande dimensão. O estado é obtido através da API de *bots* de DOTA 2 [12] e é composto por 20000 variáveis, enquanto que jogos como o Xadrez e Go possuem cerca de 70 e 400 variáveis, respetivamente [10].

O motor de jogo do DOTA 2, que é um videojogo complexo, demora cerca de 33 milissegundos a executar uma *tick*, que é o espaço de tempo utilizado para atualizar o estado de uma partida, enquanto que sistemas de Xadrez ou Go demoram nanossegundos

a efetuar o mesmo tipo de processamento, existindo uma diferença de tempo de execução na ordem de 1 milhão de vezes, entre o DOTA 2 e estes dois jogos de tabuleiro.

Foram feitas algumas alterações a variáveis do estado do jogo, durante a aprendizagem dos agentes, como tornar algo aleatórios os pontos de vida ou a velocidade do herói, de modo a forçar a exploração dos agentes. Esta metodologia resultou no começo de vitórias dos agentes contra equipas de humanos. Apesar de a OpenAI ter utilizado esta estratégia na aprendizagem, não é possível alterar, programaticamente, valores de variáveis do estado do jogo, utilizando a API de *bots* de DOTA 2.

A aprendizagem de cada agente é feita 80% contra si próprio e 20% contra anteriores iterações suas. O herói começa por explorar o mapa aleatoriamente, passando a aprender a executar *last hits*, e a estar na sua respetiva *lane*. Mais tarde o herói começa a aprender estratégias utilizadas a nível profissional, como vaguear pelo mapa para ajudar aliados a ganhar vantagem nas suas *lanes* ou juntar-se com a equipa para destruir torres adversárias.

A coordenação entre agentes foi implementada utilizando uma variável representativa do espírito de equipa, que toma valores entre 0 e 1. Sendo o valor 0, cada agente é egoísta e apenas se preocupa com a sua função de recompensa sendo o valor 1, todos os agentes agem de forma a que a função de recompensa da equipa seja o mais importante.

Foram obtidos melhores resultados, contra equipas compostas por humanos, à medida que o número de dias de treino aumentava. Começando a 15 de Maio de 2018, a equipa de agentes derrotou pela primeira vez uma equipa composta pelos melhores jogadores de DOTA 2 da OpenAI, situados no quadragésimo sexto percentil, correspondente a 2500 pontos de MMR (*Matchmaking Rating*). A 6 de Junho já conseguia vencer todas as partidas contra equipas até ao nonagésimo percentil (4000 pontos de MMR) e 4 em 6 partidas contra equipas até ao nonagésimo nono percentil (5500 pontos de MMR) [14]. Recentemente, certos intervalos de MMR foram substituídos por diferentes descrições de classificação [30].

2.4 Outras *Frameworks* para *Bots* de DOTA 2

Esta Subsecção trata de documentar o Estado da Arte no que diz respeito a implementações existentes de interfaces entre o DOTA 2 e algumas linguagens de programação, para o desenvolvimento de algoritmos de ML para problemas específicos do jogo. Nenhuma das implementações que se encontra descrita nesta Subsecção foi finalizada ou se encontra, atualmente, em desenvolvimento.

2.4.1 *Dotabots-ml-tools*

Esta implementação de interface entre o DOTA 2 e a linguagem de programação Python [31] foi desenvolvida para o SO (Sistema Operativo) Windows, não propriamente para o jogo DOTA 2, mas sim para um modo de testes e desenvolvimento de itens relacionados com o jogo, chamado DOTA 2 *Workshop Tools* [32], que não é o jogo oficial, e que tem uma API [16] mais poderosa que a API oficial de *bots* de DOTA 2 [12], não sendo portanto possível utilizar esta implementação com o jogo de DOTA 2 propriamente dito.

Neste sistema foi criada uma rede neuronal, utilizada para ensinar ao agente de DOTA 2 uma mecânica de jogo chamada *double pull*, que será explicada no parágrafo seguinte. O algoritmo foi desenvolvido em Python e trata de atualizar os parâmetros da rede neuronal, que são reescritos sobre um ficheiro LUA, que será lido através do motor de jogo do DOTA 2. Existe um *script* LUA, correspondente a um herói de DOTA2, que tratará de ler e interpretar o ficheiro com os parâmetros da rede neuronal atualizados. Com base nesta interpretação, são exeutadas as ações respetivas, a partir de um *script* LUA, que utiliza funções da API de *bots* de DOTA 2 [12] para efetuar a mecânica de jogo já mencionada.

A mecânica de jogo *pull* do DOTA 2 é utilizada para obrigar os *creeps* aliados de uma equipa (chamemos-lhe *Radiant*), de uma determinada *lane* (um dos três caminhos principais do mapa de DOTA 2), a atacarem um campo de *creeps* neutros, que se encontra na parte da floresta correspondente à equipa *Radiant*. Esta mecânica é, normalmente, realizada pelos heróis de suporte (normalmente um ou dois) da equipa *Radiant*, de modo a que se atrase a chegada dos *creeps* aliados à respetiva *lane*, obrigando os *creeps* da equipa adversária (chamemos-lhe *Dire*) a avançar ao longo da mesma *lane*, chegando mais perto da primeira torre da equipa *Radiant*. Esta mecânica pode constituir uma pequena vantagem para o *carry* (herói mais importante) dos *Radiant*, sobre o(s) herói(s) adversário(s), se os *Dire* não estiverem a pressionar muito o *carry* dos *Radiant*. Normalmente, uma simples *pull* não é suficiente para matar todos os *creeps* de uma *wave* (conjunto) de *creeps*, portanto é necessário realizar uma *double pull*, ou seja, a seguir à *pull* inicial, os heróis de suporte responsáveis pela mesma, necessitam de atrair outro conjunto de *creeps* neutros, para matarem os *creeps* restantes da *wave* dos *Radiant*.

Nesta ferramenta de *software*, o autor não implementou nenhum meio de comunicação que permita enviar comandos representativos de ações que heróis possam tomar, de forma a serem lidos por *scripts* LUA dos respetivos heróis. Apenas são lidos *logfiles* para obter o estado de partidas, de uma forma pouco eficiente, através de uma função própria, pelo menos de 100 em 100 milissegundos, devido a uma espera, na função, de 100 milissegundos, que pode não ser exata. A leitura dos ficheiros de *log* do DOTA 2 não utiliza um processo ou *thread* para paralelizar a leitura do estado com a execução de outras tarefas, levando a que a função de obtenção do estado de partidas seja chamada a cada iteração do ciclo de processamento e computação da rede neuronal. A arquitetura do sistema desta implementação, bem como o seu funcionamento, encontram-se esquematizados na Figura 2.3.

Para além das características desta implementação já enunciadas, é feita uma automatização para iniciar o ambiente da partida de DOTA 2 através da interface do utilizador.

2.4.2 *CreepBlockAI*

Semelhantemente à implementação descrita na Subsecção 2.4.1, esta implementação de interface entre DOTA 2 e um ambiente de programação (em Python) [33] foi desenvolvida com a utilização da API DOTA 2 *Workshop Tools*. Foi desenvolvida para Windows e também para Linux. O facto de esta interface, entre o jogo e um ambiente de programação em Python, utilizar a API anteriormente referida, tal como a implementação da Subsecção 2.4.1, implica que seja uma opção inválida para desenvolvimento de agentes

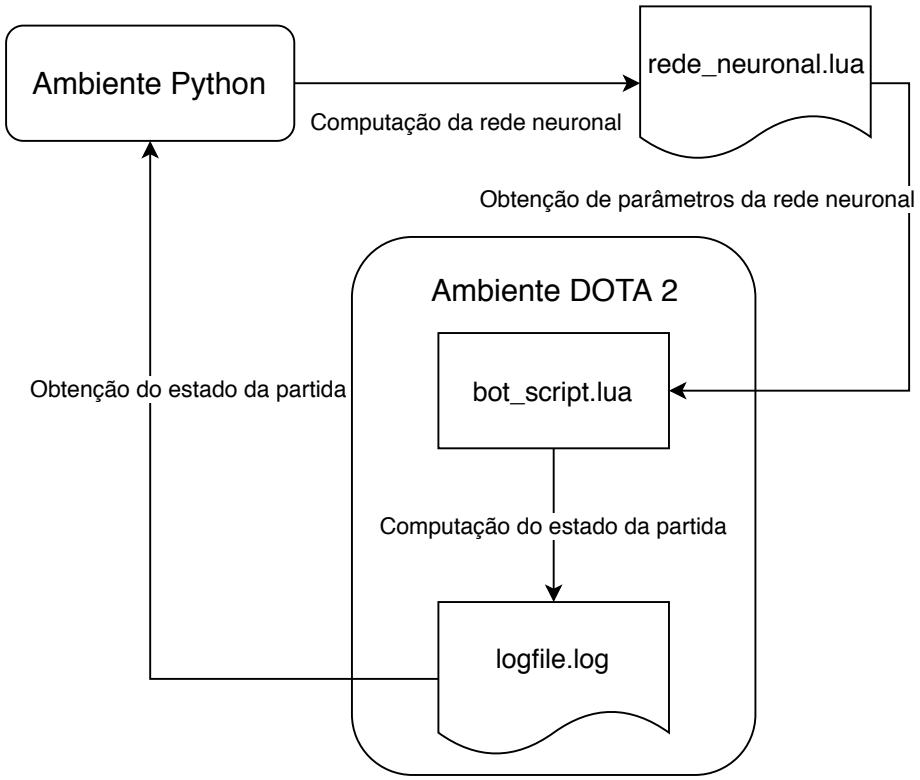


Figura 2.3: Arquitetura e lógica de funcionamento do sistema criado para implementar um agente capaz de efetuar *double pull* numa partida de DOTA 2 [31].

---

de DOTA 2, através de uma linguagem de programação que não seja LUA, pois a API DOTA 2 *Workshop Tools*, mais poderosa que a API de *scripting* de *bots* de DOTA 2, não é utilizável diretamente no jogo oficial de DOTA 2, mas sim apenas na *Workshop Tools* [32].

Esta implementação foi criada para desenvolverum agente de DOTA 2 capaz de efetuar a mecânica de jogo *blocking*. Esta mecânica consiste em bloquear ao máximo uma *wave* (conjunto) de *creeps* aliados que correm pela sua *lane*, até encontrarem os *creeps* adversários. Este bloqueio é efetuado, normalmente, por um herói, a maior parte das vezes, no início da partida, para abrandar o movimento dos *creeps* aliados, de maneira a que o confronto entre os mesmos e os *creeps* adversários ocorra na localização mais próxima possível da parte do mapa da equipa que efetua o *blocking*, ganhando, assim, vantagem territorial sobre o adversário.

O sistema desenvolvido nesta implementação [33], possui um *webservice*, desenvolvido em Python, que trata de **ler** e **atualizar** um modelo que é, inicialmente, gerado pelo DOTA 2, através de pedidos `HTTP` (*HyperText Transfer Protocol*) **GET** e **POST**, respetivamente. É obtido o último modelo do estado do jogo, ao gerar um pedido `HTTP` do tipo GET e atualiza o modelo enviando um pedido `HTTP` do tipo POST para o *webservice*, depois de ter sido processado o estado do jogo obtido e atualizado um algoritmo de ML. Analogamente, o *script* LUA desenvolvido para este sistema, **envia** e **recebe** informação através de pedidos `HTTP` do tipo **POST** e **GET**, respetivamente, recorrendo a funções da

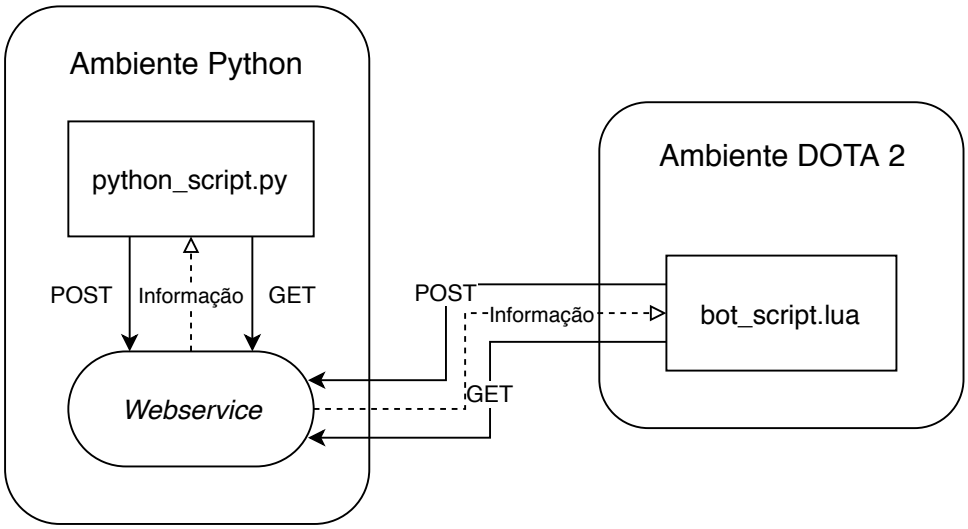


Figura 2.4: Arquitetura e lógica de funcionamento do *webservice* criado para receber informação de partidas de DOTA 2 e enviar atualizações de um algoritmo de ML que pretende ensinar a mecânica *blocking* a um agente de DOTA 2 [33].

API *Workshop Tools* de DOTA 2 [16]. Com base na informação recebida e processada, o *script* LUA do agente define uma ação para o agente executar, de modo a que este se movimente e tente bloquear os *creeps*, que correm pela *lane*, para os abrandar. Finalmente é devolvido o estado da partida para o *webservice* que retornará o modelo atualizado ao utilizador. A Figura 2.4 ilustra a arquitetura deste sistema.

O uso deste *webservice* só é possível, através de *scripts* LUA, devido ao uso da API *Workshop Tools* de DOTA 2, pelo que não é possível implementar este sistema utilizando apenas a API de *bots* de DOTA 2.

2.4.3 *Dota2comm*

Esta API [34] foi desenvolvida para Windows e para qualquer distribuição da família Unix, tendo apenas como base a linguagem de programação Python e a API oficial de *bots* de DOTA 2.

A implementação desta interface é bastante simples: permite o envio de mensagens para um *bot*, utilizando a linguagem de programação Python para escrever mensagens, codificadas, com o uso de um contador, num ficheiro LUA. Posteriormente, recorrendo à *sandbox* de LUA que o DOTA 2 fornece, a partir do *script* de um herói à escolha, é interpretado o ficheiro LUA escrito através de Python, de modo a que o herói respetivo receba a mensagem e a imprima no *chat* de equipa.

Na implementação em Windows, o autor desta API desenvolveu uma DLL (*Dynamic-Link Library*) (tal como em [17]), que permite efetuar a mesma comunicação com o jogo. Esta implementação tem a vantagem, comparativamente à implementação para Linux, de também receber mensagens do DOTA 2. O autor poderia ter utilizado a receção de

mensagens do DOTA 2 para desenvolver um *parser* do estado do jogo, visto que seria possível obter o estado, usando a API de *bots* do DOTA 2, e enviá-lo como mensagem para o utilizador.

2.4.4 *Dota2-WebAI*

Esta API [35] foi criada com o intuito de fornecer, ao seu utilizador, o controlo de um agente de DOTA 2, programaticamente, mas a partir de um ponto de vista de alto nível. Isto significa que o utilizador desta interface não codificará cada ação do *bot* com base no estado do jogo, mas sim dará ordens predefinidas, pouco detalhadas de ações que o agente pode tomar, mas que são muito comuns numa partida deste videojogo. Este tipo de ações pode ser denominado de "abstrato". Um exemplo de uma ação abstrata é definir uma ação para o agente, de modo a que se desloque para a *lane* ou "caminho" da parte de baixo do mapa (*bottom lane*) e que mate os *creeps* adversários que encontrar [35]. A esta mecânica de jogo dá-se o nome de *farming*.

A arquitetura desta interface, à semelhança de [33] (Subsecção 2.4.2) baseia-se no uso de um servidor *backend* para enviar e receber pedidos `HTTP` para o DOTA 2 e vice-versa. A comunicação é estabelecida através de objetos JSON entre este servidor local e o motor de jogo bem como a API DOTA 2 *Workshop Tools* e a API oficial de *bots* de DOTA 2. Só é possível estabelecer a comunicação desta forma utilizando a API DOTA 2 *Workshop Tools*; não é possível fazê-lo com a API oficial de *bots* de DOTA 2.

O modo de implementação deste sistema é *Complete Takeover* [12], ou seja, o autor apenas controla a função `Think()` na implementação de *scripts* em LUA; não controla funções como `GetDesire()`. Esta função tem o objetivo de consultar o motor de jogo do DOTA 2 de forma a averiguar qual é o desejo de continuar num determinado modo de implementação. Isto retira todo o processo de decisão numa implementação de ML para um agente, pois o motor de jogo é que, com o uso desta função, faz a avaliação do estado do jogo. No entanto, o autor em [35] propõe um sistema que possibilite ao utilizador do mesmo um controlo de mais alto nível. Isto sugere que o utilizador desta interface não teria acesso aos métodos da API de *bots* diretamente, mas sim a ações mais abstratas, fornecidas por esta interface, baseada num servidor local para controlar um agente de DOTA 2.

A atualização do estado do jogo encontra-se dividida em quatro componentes:

- Informação sobre o estado das partidas, que é composto por:
  - Informação sobre o estafeta da equipa do *bot*, nomeadamente os seus pontos de vida, se é voador, o seu estado e a sua localização dada por coordenadas X, Y e Z:
  - Informação sobre a quantidade de *creeps*, em cada uma das *lanes* ou "caminhos" do mapa, tanto da sua equipa como da equipa adversária.
- Informação sobre o estado do herói atualmente a ser controlado através desta interface, nomeadamente o seu nome, equipa, pontos de vida, de *mana*, de vida máxima

e de *mana* máxima, bem como a percentagem de regeneração de vida e de *mana*, rapidez de movimento, localização, itens e o estado dos mesmos, o estado das suas habilidades, ouro e pontos de experiência.

- Informação sobre o estado dos aliados do herói controlado. O estado dos aliados é composto pelos mesmos atributos do estado do herói, exceto o estado dos itens e o estado das habilidades.

- Informação sobre o estado dos inimigos do herói controlado. O estado dos inimigos é composto pelos mesmos atributos do estado dos aliados do herói controlado, exceto o ouro e pontos de experiência, pois numa partida de DOTA 2 não é possível saber estas últimas duas informações sobre heróis da equipa adversária.

Estas quatro componentes do estado do jogo não são atualizadas em intervalos de tempo iguais. As atualizações dependem do processamento de pacotes (objetos JSON) ou, no máximo, de um intervalo de tempo. Na atualização do estado do mundo, o intervalo de tempo é de 0.5 segundos. Na atualização do estado do herói controlado, o intervalo de tempo é de 0.25 segundos. Na atualização do estado dos aliados do herói controlado, o intervalo de tempo é de 0.5 segundos. Na atualização do estado dos inimigos do herói controlado, o intervalo de tempo é de 0.25 segundos. É possível, assim, concluir que a atualização do estado do jogo não é atómica, o que pode levar a inconsistências no processamento do estado do jogo do lado do cliente desta API. A obtenção do estado do jogo nesta implementação tem uma taxa de atualização de 4 Hz e 2 Hz, dependendo do que tipo de atualização, que é 7.5 e 15 vezes mais lento do que a taxa de atualização de estados no motor de jogo do DOTA 2 (30 Hz), como demonstrado na Secção 3.3 desta dissertação.

2.4.5 Outras Implementações

Dota2 AI Framework [36] é uma ferramenta de *software*, implementada em Java, que se baseia, tal como as implementações [35] e [33], num *webserver* que processa pedidos `HTTP` para estabelecer comunicação entre o DOTA 2 e um ambiente de programação, que possibilite o desenvolvimento de agentes para este videojogo, neste caso, recorrendo à linguagem de programação Java. Devido à utilização de um servidor *web* para estabelecer a ligação entre a *framework* e o jogo, conclui-se que esta implementação, tal como a maior parte das implementações já descritas nesta Subsecção, não é útil para o desenvolvimento de agentes de DOTA 2, pois recorre à API da *Workshop Tools* do DOTA 2 para o processamento dos pedidos `HTTP`, enviados através da *framework*, bem como para a mesma.

A implementação *Dota2DQN* [37] foi efetuada em Windows, com recurso ao método de comunicação implementado em [34] e à linguagem de programação Python, para estabelecer a comunicação entre um algoritmo de ML (que o autor não explica para que finalidade é utilizado) e o estado do jogo. No repositório não existem ficheiros LUA, pelo que não é possível determinar como o autor gera e envia o estado do jogo a partir de um *script* de um agente, em LUA, de modo a utilizá-lo no seu código, onde criou um algoritmo de ML,

em Python. Não existe qualquer tipo de automatização para a criação de ambientes de partidas. É possível deduzir que, devido ao uso de comunicação presente em [34], nesta implementação não existe necessidade de utilizar a API *Workshop Tools* do DOTA 2. No entanto esta dedução pode não ser eficaz pois não foram fornecidos elementos que comprovem a utilização de *scripts* LUA para o estabelecimento da comunicação (receção de estados de partidas e envio de comandos representativos de ações que agentes de DOTA 2 possam tomar).

Por fim, o *Dota2-FullOverwrite* [38] não é uma interface entre o jogo e um ambiente de programação com uma linguagem diferente de LUA, mas sim uma tentativa de reestruturação, através da API de *bots* de DOTA 2, da codificação de *bots* deste jogo. Com esta implementação, pretende-se fornecer aos utilizadores da mesma uma abordagem de mais alto nível ao *scripting* de todos os heróis de DOTA 2, reescrevendo a utilização de itens e habilidades especiais, bem como a implementação de mecânicas de equipa para ser possível estabelecer maior sinergia entre os seus elementos.

2.5 Interfaces para *Bots* noutros Videojogos

Nas seguintes Subsecções encontram-se descritas as interfaces criadas para *bots* noutros videojogos, nomeadamente o videojogo UT, em que foi utilizado o sistema *Gamebots* (Subsecção 2.5.1), e o videojogo Quake II, em que foi utilizado o sistema *Soar* (Subsecção 2.5.2).

2.5.1 *Gamebots*

UT [39] é um jogo do género FPS (First-Person Shooter), ou em português, atirador em perspetiva de primeira pessoa. Este jogo consiste em matar adversários com recurso a diferentes e variadas armas que são apanhadas no mapa durante o decorrer da partida. Existem três modos de jogo principais, *Deathmatch*, *Domination* e *Capture the Flag*. Os autores implementaram o módulo *Gamebots* [17] de modo a cobrir todos estes modos de jogo, caracterizando-os como tarefas.

De uma maneira análoga ao DOTA 2, o UT fornece uma linguagem de programação de *scripting*, de nome UnrealScript, para desenvolver novos modos de jogo. O UnrealScript foi desenvolvido em C++ [17], tal como a linguagem de programação LUA foi desenvolvida em C [40].

O *Gamebots* [17] é um módulo criado para UT que apresenta uma arquitetura baseada em *sockets* numa rede local, utilizadas para transportar informação protocolar. A informação enviada e recebida pelas *sockets* consiste em mensagens constituídas por apenas uma linha. As *sockets* estabelecem a comunicação entre um cliente e um servidor, podendo ser o cliente um *bot* ou um humano e o servidor ser o *Gamebots*, a que os *bots* se conectam, ou o servidor de UT, a que os humanos se conectam. O módulo fornece informação do estado do jogo às personagens, para que estas possam tomar ações. As ações são convertidas em comandos, que são enviados através de *sockets*, desta vez de volta para

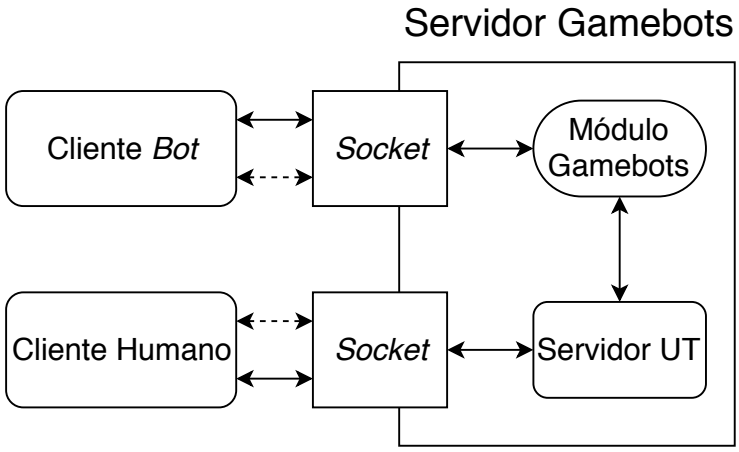


Figura 2.5: Arquitetura do *Gamebots* [17]. As setas de linha cheia e linha a tracejado, entre as *sockets* e os clientes, representam a comunicação periódica e aperiódica, respetivamente. As outras setas representam o fluxo de informação do sistema. O cliente humano liga-se diretamente ao servidor UT e o cliente *bot* liga-se, primeiramente ao módulo *Gamebots*, que, por sua vez, se liga ao servidor UT.

---

o servidor do módulo *Gamebots*. A Figura 2.5, adaptada de [17], ilustra a arquitetura do *Gamebots*.

A comunicação entre *sockets* é constituída por três fases, o *handshake*, comunicação periódica e comunicação aperiódica, como ilustra a Figura 2.6, mais detalhadamente. Na primeira fase é feita a comunicação inicial entre *sockets*, de modo a estabelecer a ligação entre as mesmas. A segunda e terceira fases compõem o envio de mensagens, que contêm, o que os autores de [17] determinam como, "informação sensorial". As mensagens periódica são enviadas dez vezes por segundo e as aperiódicas são enviadas quando certos eventos ocorrem em intervalos de tempo em que é efetuado o envio das mensagens periódicas. Um exemplo de mensagens aperiódicas é a troca de mensagens entre utilizadores humanos. O conteúdo das mensagens da fase periódica é constituído por atributos da partida, nomeadamente o resultado ou o tempo de jogo. Para além destes atributos, as mensagens periódicas incluem toda a informação que um jogador humano teria no seu campo de visão em determinado momento do jogo, como por exemplo as armas que possui ou os adversários visíveis. As mensagens da fase aperiódica são enviadas como atualizações da informação sensorial. Um exemplo destas atualizações é a personagem ter ouvido um som.

Neste sistema, o agente não tem perceção do que o rodeia, pelo que a sua posição no mapa é determinada por coordenadas de localização (x, y e z) e de rotação (em graus), sendo a arquitetura do mapa constituída por obstáculos alcançáveis e não alcançáveis ao agente. Os autores do módulo *Gamebots* afirmam que com este conjunto de coordenadas é possível fazer o *bot* navegar a maior parte dos mapas de UT [17].

Como referido anteriormente, o envio de comandos para o agente é realizado através das *sockets* da rede local para o módulo *Gamebots*. São exemplos de comandos as seguintes

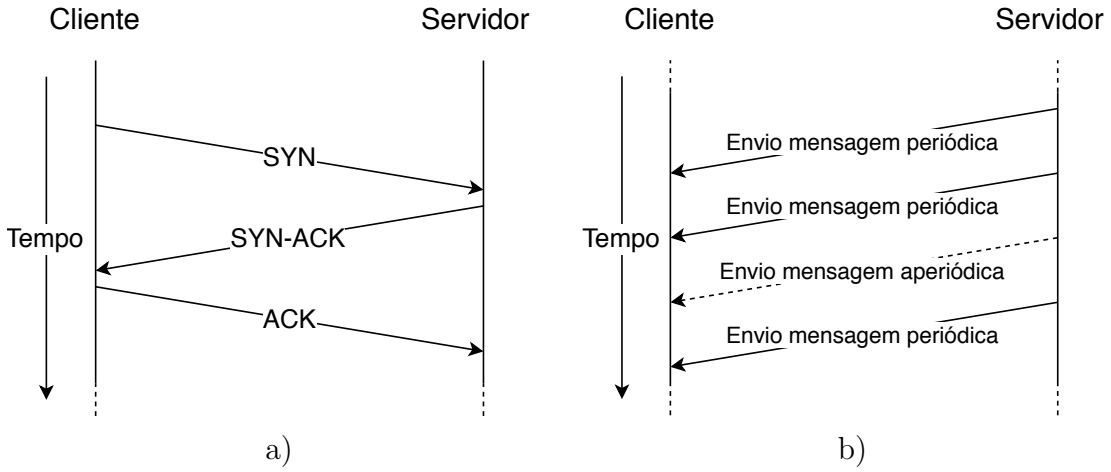


Figura 2.6: Comunicação entre cliente e servidor, no sistema *Gamebots*. a) *Handshake* entre cliente e servidor, para estabelecimento de comunicação. b) Comunicação periódica e aperiódica entre cliente e servidor, seguidamente ao *handshake* inicial.

---

ações:

- *STOP*: força o agente a parar.
- *JUMP*: força o agente a saltar.
- *RUNTO*: força o agente a caminhar até um alvo ou até uma localização determinada por coordenadas.
- *CHANGEWEAPON*: força o agente a mudar para a arma com o nome do argumento do comando.

Este sistema introduziu a possibilidade, inovadora para a altura, de os agentes controlados por *software* interagirem com jogadores humanos e jogarem em conjunto com os mesmos num ambiente comum.

No sistema *Gamebots*, personagens do UT controladas por humanos podem comunicar entre si através de mensagens globais ou de equipa. Estas mensagens são aperiódicas e serão recebidas por qualquer agente que esteja disposto a recebê-las. Com o uso do deste módulo, um humano tem a possibilidade de interagir com *bots* da sua equipa através mensagens predefinidas como "*Defendam a base*", "*Mantenham as vossas posições*" ou "*Cubram-me*".

2.5.2 *Quakebot*

Quake II [41], mais concretamente o modo de jogo *Deathmatch*, similarmente ao modo de jogo *Deathmatch* do UT [39], é composto por um mapa onde os jogadores podem apanhar armas e objetos que lhes dão poderes (aos jogadores). Este modo de jogo tem como objetivo, para o jogador, matar os inimigos o mais rapidamente possível até chegar

ao número limite de mortes. Cada vez que o jogador é atingido por um inimigo perde pontos de vida e armadura (se tiver). Quando os pontos de vida são nulos a personagem morre e voltará a renascer num dos locais determinados para o efeito. Este é o único modo de jogo do Quake II para o qual foi feita uma implementação do ***Quakebot***, com recurso ao sistema *Soar* [42].

De modo a entender a implementação da interface entre o videojogo Quake II e o controlo de personagens do mesmo é necessário ter a noção da composição do *Soar*. Este sistema foi utilizado para criar um *bot* de Quake II que tem implementada a antecipação de localização de inimigos [18].

Arquitetura do Sistema *Soar*

Soar [42] é um sistema desenhado com o objetivo de resolução de **problemas**. De seguida encontra-se descrito o modo de funcionamento do *Soar* em termos de funções base do sistema.

Existem dois tipos de funções que compõem a base desta arquitetura:

- **funções de implementação de tarefa**: têm o objetivo de gerar e obter "***problem spaces***" (**domínios do problema**) [42], **operadores** e **estados**.
- **funções de controlo de pesquisa**: têm o objetivo de encontrar domínios do problema, um estado que esteja disponível, e um operador.

O problema principal a resolver com o uso do *Soar* é dividido em domínios de problemas. Estes domínios de problemas são provenientes de uma função de implementação de tarefa. Recorrendo a uma função de controlo de pesquisa, é selecionado um **domínio de problema** e, de seguida, é selecionado um estado, o **estado inicial**, a partir do conjunto de estados possíveis, provenientes da função de implementação de tarefa. Finalmente é selecionado um **operador**, também ele proveniente da função de implementação de tarefa, a utilizar sobre o estado previamente escolhido. Os estados consequentes surgem da ação dos operadores sobre o(s) estado(s) anterior(es). Existem outras funções importantes que têm como objetivo a **criação**, **seleção** e **finalização de objetivos**, **gestão de memória** e **aprendizagem**, não incluídas no processo de implementação de tarefa nem no controlo de pesquisa.

Seguidamente encontram-se os componentes (Figura 2.7) que constituem a arquitetura do sistema *Soar*, bem como a descrição dos mesmos:

- ***working memory*** ou **memória de trabalho**: tem o objetivo de processar o estado atual e é composta por três componentes:
  - ***context stack*** ou **pilha de contexto**: tem o objetivo de determinar a "hierarquia dos objetivos atuais, domínios dos problemas, estados e operadores" [42];
  - ***objects*** ou **objetos**, compostos pelos objetivos atuais a atingir em determinados contextos e os estados respetivos aos contextos;

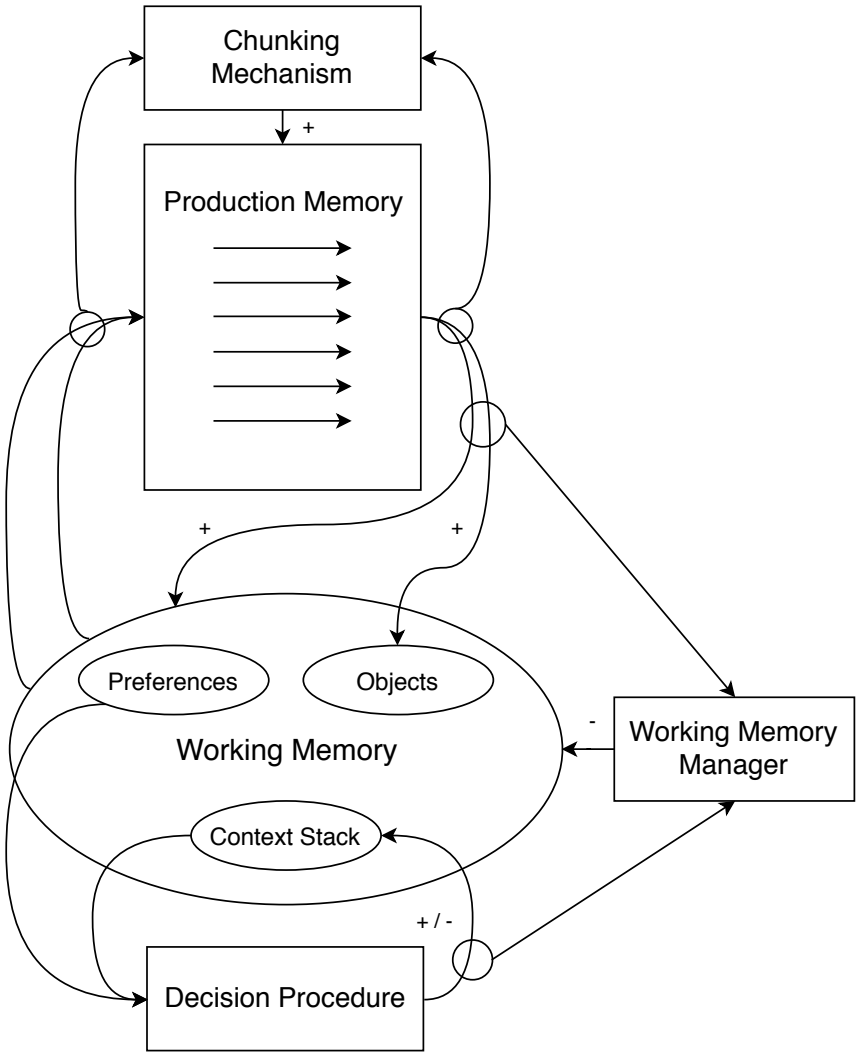


Figura 2.7: Diagrama de Arquitetura do *Soar* [42].

  - ***preferences*** ou **preferências**: têm o objetivo de codificar a aprendizagem obtida através do controlo da pesquisa.

- ***production memory*** ou **memória de produção**: composta por um conjunto de produções que adicionam preferências e objetos à memória de trabalho;

- ***decision procedure*** ou **procedimento de decisão**: tem o objetivo de, com base nas preferências e na pilha de contexto, poder alterar esta última, adicionando-lhe ou retirando-lhe elementos;

- ***working memory manager*** ou **gestor da memória de trabalho**: tem o objetivo de eliminar elementos da memória de trabalho, para uma gestão eficiente da memória do sistema;

- ***chunking mechanism*** ou **mecanismo de aglomeração**: tem o objetivo de adicionar produções à memória de produção, sendo responsável pela aprendizagem do sistema [42].

O termo *chunking* provém da psicologia. George Miller em 1956 definiu o termo *chunking* como a "organização e aglomeração de uma entrada de informação em *chunks*", ou pedaços [43].

Utilizando uma **função de implementação de tarefa**, o *Soar* gera **estados** e **operadores**. Seguidamente o sistema seleciona e aplica operadores a estados, utilizando uma **função de controlo de pesquisa**, no denominado "**ciclo de decisão**". Este ciclo é composto pela "**fase de elaboração**" e pelo "**processo de decisão**". Na fase de elaboração, novos **objetos** (**objetivos** e estados) são adicionados e/ou alterados e novas **preferência**s são adicionadas à **memória de trabalho**, recorrendo à **memória de produção** (Figura 2.7. O processo de decisão tem o objetivo de examinar as preferências adicionadas para o operador e, assim, substituir um objeto existente ou criar um novo **subobjetivo** se ocorrer um **impasse**. Um impasse ocorre quando o processo de decisão do *Soar* não consegue terminar, devido à falta de conhecimento sobre como proceder em determinado ponto deste processo. Assim, a arquitetura do *Soar* cria subobjetivos, de modo a resolver o impasse. Esta operação tem o nome de *criação automática de subobjetivos* [42].

Sistema *Soar* Aplicado a ML para o Videojogo Quake II

O Quakebot, ao contrário do sistema *Gamebots* [17], apenas fornece a possibilidade de controlar uma personagem.

Similarmente ao sistema *Gamebots*, a informação obtida do jogo, bem como os comandos possíveis para o agente, são idênticos aos que um jogador teria numa partida. Tal como no *Gamebots*, o Quakebot constrói uma perceção do mapa através da distância da personagem aos elementos do mesmo. Este mapa fica guardado para não ter de ser feita uma aprendizagem do mesmo cada vez que o Quakebot é executado.

A interface entre o jogo e o *Soar* é realizada através de uma DLL que contém código escrito em C para ligar o Quakebot à informação obtida do jogo bem como o envio de comandos de ações do *bot*. Para estabelecer esta ligação foram criadas *sockets* para comunicação entre computadores, visto que o Quakebot foi executado numa máquina diferente da que executava o videojogo, nesta implementação [18].

O motor de jogo atualiza o estado da partida dez vezes por segundo, ao invocar métodos da DLL desenvolvida responsáveis pela atualização de estado, a cada décima de segundo. Quaisquer ações que o *Soar* tenha definido são efetuadas na próxima atualização de estado. O sistema *Soar* executa o ciclo de decisão entre trinta a cinquenta vezes por segundo de um modo assíncrono ao videojogo, podendo alterar decisões de execução, no máximo, cinco vezes por atualização de estado.

Como explicado na Subsecção 2.5.2, um **operador**, parte integrante da **memória de trabalho**, mais concretamente da **pilha de contexto** (Figura 2.7) atua sobre um **estado**, também este integrante dos **objetos** da **memória de trabalho**, originado um novo estado.

Nesta API um operador é um comando representativo de uma ação que se pretende que o agente execute. As ações (operadores) propostas para aplicar ao Quakebot dividem-se em três tipos: ações básicas, como por exemplo mover ou disparar, ações internas ao *bot*, nomeadamente lembrar-se da última posição de um inimigo, e ações abstratas, mais complexas e que são compostas por conjuntos de ações. Um bom exemplo de uma ação abstrata é apanhar um item.

Ao aplicar um operador sobre um estado, é gerado um novo estado e existe, assim, a possibilidade de gerar subestados, estados "filho", a partir de um estado anterior, o estado "pai". Tomando o exemplo já mencionado de ação abstrata "apanhar_item", podemos considerar que esta envolve uma árvore de suboperadores, constituída por "ir_ate_item", "encarar_item" e "mover_ate_item". Interpretando a Figura 2.8, que ilustra este exemplo, é possível verificar que, para executar a ação "apanhar_item", é necessário dividir o operador em suboperadores. Assim sendo, tem-se o suboperador de "apanhar_item", chamado "ir_ate_item" e os suboperadores do suboperador "ir_ate_item", chamados "encarar_item" e "mover_ate_item". Portanto, para executar a ação abstrata de apanhar um item é necessária a ação de ir até ao item, que é executada se for executada a ação de mover até ao item, se o *bot* já estiver direcionado para o item, caso contrário necessita de encarar o mesmo e depois mover-se até ele.

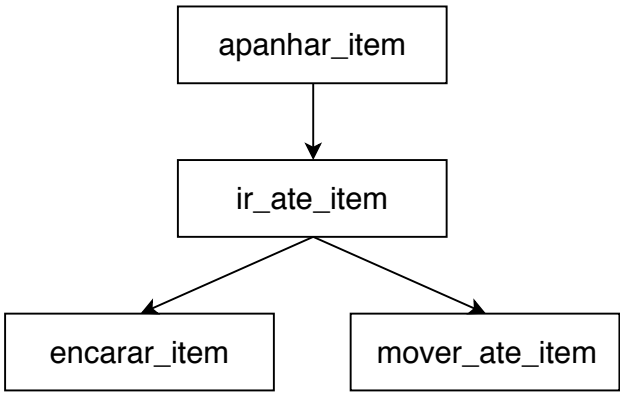


Figura 2.8: Árvore de operadores utilizados para executar a ação abstrata "apanhar_item".

No exemplo do **operador** "apanhar_item", no **processo de elaboração** do **ciclo de desisão** do *Soar*, são criadas **preferências**, para o operador, que fazem parte da **memória de trabalho**, nomeadamente a distância do *bot* até ao item ou a relevância deste no **estado** atual. Seguidamente, no **processo de decisão** do ciclo de decisão, é escolhido o melhor operador para o estado atual e é criado um novo comando, "ir_ate_item". Para melhor perceber este exemplo, o leitor deve ver a Figura 2.7 que representa a arquitetura do *Soar* e a Secção 2.5.2 deste Capítulo, que explica o funcionamento do ciclo de desisão da arquitetura.

2.6 Conclusão

Neste Capítulo foi feita uma descrição sobre o jogo de DOTA 2, onde foi explicado o funcionamento do mapa, das unidades e das partidas deste jogo. Foi também abordado, neste Capítulo, a API oficial de *scripting* de *bots* de DOTA 2, nomeadamente os seus tipos de implementação e funções constituintes. Foram descritas duas implementações de agentes de DOTA 2, nomeadamente a implementação de apenas um agente e outra implementação multiagente, criadas e desenvolvidas pela OpenAI 2.3. Foi também descrito o estudo efetuado sobre implementações existentes de *frameworks* para o DOTA 2, para o desenvolvimento de agentes deste jogo. Na parte final deste Capítulo foram estudados dois sistemas que permitem o controlo programático de agentes, em dois videojogos (UT e Quake II), de forma a criar IAs para os mesmos, tendo o sistema criado, para Quake II, uma arquitetura que permite o desenvolvimento de agentes inteligentes não sendo necessário utilizar ML.

Nenhuma das implementações existentes de *frameworks* para agentes de DOTA 2 estudadas se encontra concluída ou em desenvolvimento, nem apresenta uma proposta de solução relevante para o desenvolvimento de uma interface para o videojogo. Algumas destas implementações nem permitem o envio de comandos representativos de ações para os heróis executarem ou mesmo a leitura do estado de partidas de uma forma eficiente.

No caso do *Gamebots* (Secção 2.5.1), é possível que os utilizadores joguem contra implementações de IAs criadas através deste sistema no videojogo UT, tal como no DOTA 2, em que os utilizadores têm a possibilidade de jogar contra *bots* desenvolvidos através de *scripts* LUA, num *lobby* local.

Capítulo 3

O Sistema Desenvolvido

Neste Capítulo encontra-se a descrição da arquitetura do sistema desenvolvido nesta tese (Secção 3.1) e do fluxo de informação no mesmo sistema (Secção 3.2). Na Secção 3.3 encontra-se descrita a implementação do *script* LUA do herói *Ursa* que pode ser aplicado a qualquer herói de DOTA 2, e na Secção 3.4 encontra-se a implementação do módulo desenvolvido em Python, denominado `PythonDota2`, que tem o propósito de fornecer, ao seu utilizador, uma interface de alto nível para o jogo DOTA 2, mais concretamente para a API de *bots* de DOTA 2. Ainda nesta Secção estão descritas as funções disponibilizadas no módulo, representativas de ações que heróis de DOTA 2 possam tomar, bem como a explicação da diferença entre implementações de paralelização de uma função de leitura de *logfiles* com o processo principal da execução do módulo, entre processos e *threads* e a decisão sobre a escolha entre as duas implementações, para a implementação final.

Neste trabalho foi efetuada uma implementação, para a família de SOs Unix, que envolve uma interface entre o jogo DOTA 2 e o módulo desenvolvido em Python. Esta *framework* foi criada, de raiz, sob a forma de um módulo denominado `PythonDota2`, para ser utilizado em Python. A implementação de *scripts* LUA para *bots* de DOTA 2 é efetuada recorrendo à API de *bots* de DOTA 2, que apresenta três conjuntos extensos de funções, referidos no Capítulo 2. O DOTA 2 fornece uma interface entre o jogo e uma *sandbox* em LUA, para programar *scripts* para *bots* deste jogo, que podem ser utilizados em *lobbies* locais. A *sandbox* fornecida para programação de *scripts* LUA limita, por motivos de segurança, algumas funcionalidades, nomeadamente, importar bibliotecas externas e impedir o uso de bibliotecas padrão de LUA, como é o caso da biblioteca `io` [44]. Neste sistema desenvolvido, foi criado um *script* para o herói *Ursa* [45], que comunica com um módulo desenvolvido em Python, de modo a possibilitar a receção de comandos representativos de ações para o herói executar (enviados através do módulo), bem como a recuperação de informação sobre o estado das partidas, a partir da perspetiva da linguagem de programação Python. Como também referido no Capítulo 2, o modo de implementação do *script*, para o herói *Ursa*, é o modo *Complete Takeover*. Este modo de implementação é o único que poderia fornecer, ao utilizador deste sistema, um controlo total sobre o herói. Apesar do controlo total sobre o herói, a interface criada durante este trabalho é de alto nível, similar à API de *bots* de DOTA 2. Assim sendo, o módulo `PythonDota2` pode ser considerado

uma API, que permite controlar *bots*, em Python, para a API oficial, que permite controlar *bots*, em LUA.

3.1 Arquitetura do Sistema

A principal razão para a elaboração deste trabalho foi o facto de, como já referido no Capítulo 2, atualmente não existirem APIs, para DOTA 2, que forneçam uma interface entre a API de *bots* oficial do jogo e uma linguagem de programação como Python ou C++. Por não existir tal API não é possível utilizar ML, utilizando linguagens de programação mais poderosas que LUA, para desenvolver um agente inteligente para o DOTA 2.

A definição da interface entre o jogo DOTA 2 e a linguagem de programação Python foi executada tendo em conta as limitações da *sandbox* que a API de *bots* oficial de DOTA 2 apresenta. Uma das limitações é o facto de ser impossível importar bibliotecas ou usar bibliotecas padrão de LUA num *script* dentro da *sandbox* fornecida pelo jogo, para programar *bots*. Assim sendo, não é possível ler ficheiros nem escrever novos ficheiros, usando a API de *bots*, pois a biblioteca nativa de LUA não o permite, porque apenas é possível fazê-lo usando a biblioteca padrão `io` [44], que a *sandbox* que o DOTA 2 impede de utilizar. Neste Capítulo é detalhada a forma usada para contornar este problema.

Apesar do problema descrito no parágrafo anterior, foi possível ler ficheiros LUA a partir do *script* do herói que se pretende controlar através de *software*. Para além deste problema, também não é possível escrever conteúdo para ficheiros a partir do mesmo *script*, no entanto é possível imprimir o estado do jogo, obtido através da API de *bots*, para um ficheiro *log* (ficheiro de *output* de informação imprimida no terminal do DOTA 2). Este ficheiro é, então, lido com recurso ao módulo Python desenvolvido, e todas as informações do estado do jogo, obtidas a partir do *logfile*, são tratadas e atualizam variáveis do estado do jogo, presentes no módulo.

O sistema é composto pelo módulo `PythonDota2`, desenvolvido nesta tese, com recurso à linguagem de programação Python, bem como o *script* LUA para o herói *Ursa*, também desenvolvido nesta tese, que trata de comunicar com o módulo `PythonDota2`. A Figura 3.1 representa a interface desenvolvida entre o módulo `PythonDota2` e o jogo.

O DOTA 2, quando iniciado pela primeira vez, escreve os seus *logs* para um ficheiro chamado `console.log` e, se for iniciada uma nova partida, através de um *lobby* local, o *logging* é interrompido no ficheiro `console.log` e é iniciado num ficheiro que contém o ID (Identificador) da nova partida no seu nome, sendo o nome do novo *logfile* do tipo `console.<id_partida>.log`. Futuramente, quando terminada a partida, sem o processo de DOTA 2 ser terminado, o *logging* continuará a ser executado para o *logfile* anteriormente criado com o ID da partida que agora terminou. Quando for criada uma nova partida, será repetido o processo já descrito.

O jogo, ao ser iniciado através do módulo `PythonDota2`, lê automaticamente o ficheiro `autoexec.cfg`, que contém duas configurações e que é criado pelo **Processo Principal**. Uma permite o uso de *cheats* (códigos), que apenas servirão para efetuar a criação do ambiente de jogo, não influenciando os acontecimentos das partidas. A outra configuração

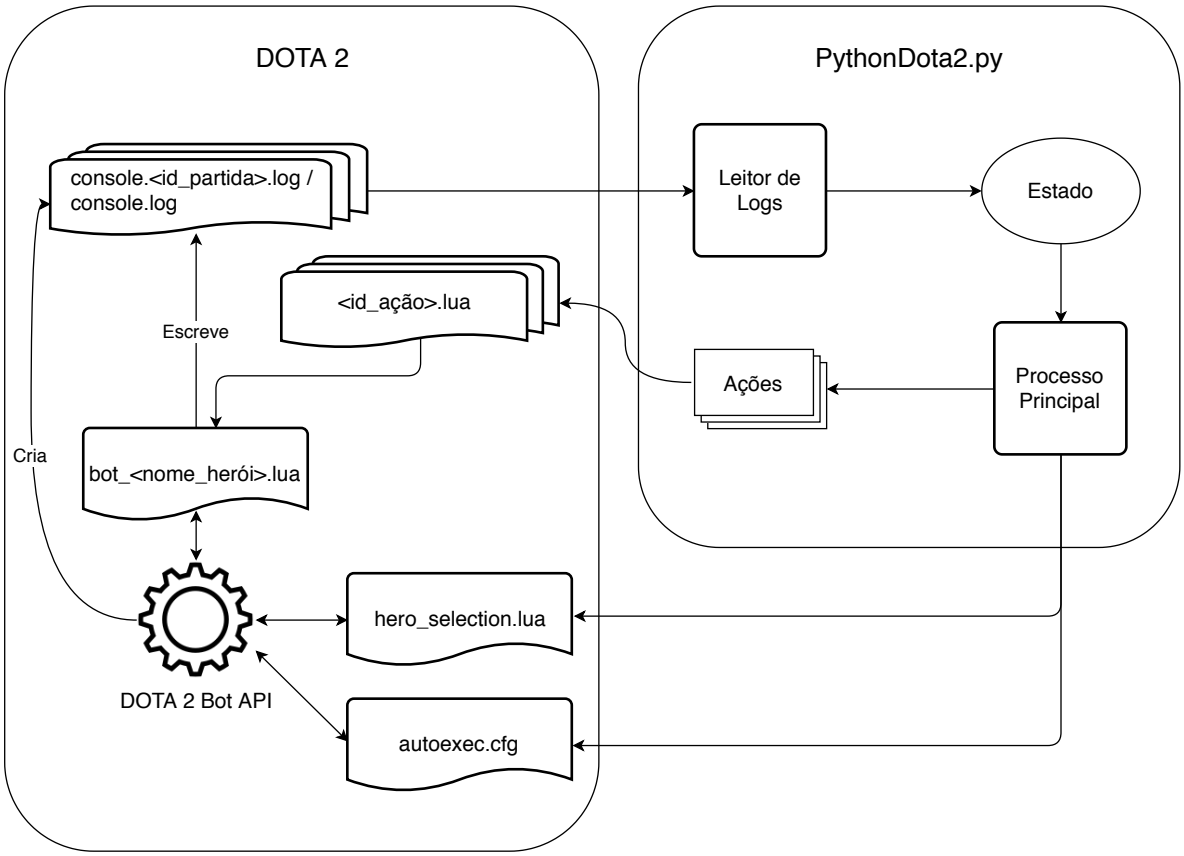


Figura 3.1: Diagrama de Arquitetura do Sistema.

serve apenas para que não sejam imprimidos avisos, que encheriam o *logfile* em demasia. Os avisos nada têm a ver com a partida, mas sim com o Steam [46], que é a plataforma de distribuição deste jogo.

Ao iniciar uma partida, usando o módulo desenvolvido, é inicializado o **Leitor de *Logs*** que irá efetuar *parsing* dos *logfiles* do DOTA 2, retirando informação estruturada do estado da partida, presente nesses ficheiros. Esta informação será atribuída a variáveis do **Estado** do módulo `PythonDota2`, a uma frequência de trinta vezes por segundo, que é o mesmo número de vezes por segundo que o motor de jogo do DOTA 2 é atualizado (trinta *ticks* por segundo), fazendo progredir a partida. Seguidamente é efetuada uma automatização através da GUI do jogo, para criar a partida com o modo de jogo escolhido através do módulo, em que será controlado o herói pretendido, na equipa escolhida e que defrontará um adversário também escolhido pelo utilizador do `PythonDota2`. De modo a controlar um herói é necessário ter um *script* LUA para o herói respetivo, que estabeleça a comunicação com o módulo Python, que pode ser igual ao *script* criado para o herói *Ursa*, para este sistema. O herói adversário não necessita de um *script* LUA. O utilizador do módulo, ao escolher estes dois heróis e a equipa do herói que pretende controlar, fará com que o ficheiro `hero_selection.lua` seja alterado, de acordo com as suas escolhas, através do Processo

Principal. Depois de alterado, este ficheiro é lido pela API de *bots*, ao criar o *lobby* da partida e, seguidamente, esta começará.

Utilizando do módulo desenvolvido, as variáveis do Estado são automaticamente atualizadas mas, para serem acedidas pelo utilizador, este necessita de invocar as funções correspondentes a cada variável do estado do jogo, funções essas, que são fornecidas pelo módulo `PythonDota2`. Com base nas variáveis do Estado obtidas, o agente desenvolvido em Python decide que comandos (**Ações**) serão enviados ao herói que se pretende controlar. Estas ações, que têm IDs associados, são escritas sob forma de um bloco de código LUA, que retorna uma *string*, para um ficheiro LUA, que terá o ID da ação no seu nome. Estes ficheiros LUA terão um nome do tipo `<id_ação>.lua`.

De volta ao DOTA 2, no *script*, denominado `bot_<nome_herói>.lua`, do herói que se pretende controlar, é executada a função presente no ficheiro `<id_ação>.lua`, que foi a ação previamente definida a partir do Processo Principal, em Python. Se a *string* retornada for correspondente a uma ação, esta é executada, utilizando a API de *bots* do DOTA 2.

3.2 Fluxo de Informação do Sistema

O Fluxo de Informação do Sistema, como ilustra a Figura 3.2, engloba toda a circulação de informação no sistema criado, envolvendo o jogo DOTA 2, a API de *bots* do jogo e o módulo `PythonDota2`. Para além do fluxo de informação no sistema, o diagrama de fluxo de informação demonstra todo o processo de decisão e lógica que o sistema possui. Estes processos advêm tanto do *script* criado, como do DOTA 2, que tem a sua própria lógica de funcionamento, que não é possível alterar.

A lógica do sistema começa quando um *script* criado em Python, que utiliza funções do módulo `PythonDota2`, é executado. No início da sua execução, é são lidas, pelo módulo *PythonDota2*, as variáveis responsáveis pela criação do ambiente de jogo e do estado da partida e é editado o ficheiro `hero_selection.lua`, de acordo com o herói a controlar, equipa e herói adversário escolhidos.

Paralelamente irão existir três Processos, que podemos denominar de ***Processo 1***, ***Processo 2*** e ***Processo 3***.

O *Processo 1* (Algoritmo 1) trata de iniciar o ambiente da partida de DOTA 2. De modo a iniciar o ambiente da partida é necessário iniciar o DOTA 2, se este não se encontrar já iniciado, e efetuar uma automatização através da GUI do jogo, de modo a criar um *lobby* para a nova partida com o modo de jogo pretendido e com o ficheiro `hero_selection.lua`, previamente alterado. Uma vez iniciada a partida, é lido o *script* LUA, do herói que se pretende controlar e obtida a informação sobre o estado do jogo, que é imprimida para o *logfile* `console.<id_nova_partida>.log`. Por fim, é lido o ficheiro `<id_comando>.lua` que pode conter algum comando para ser executado. O mesmo é carregado, através do *script* do herói e, se o comando existir na API de *bots* de DOTA 2, ele será executado, verificando depois se a partida terminou, caso contrário a verificação do fim da partida é imediata. Se a partida realmente terminou, é concluído o ciclo geral do sistema, caso contrário o *script* LUA do herói é lido de novo e este ciclo repete-se.

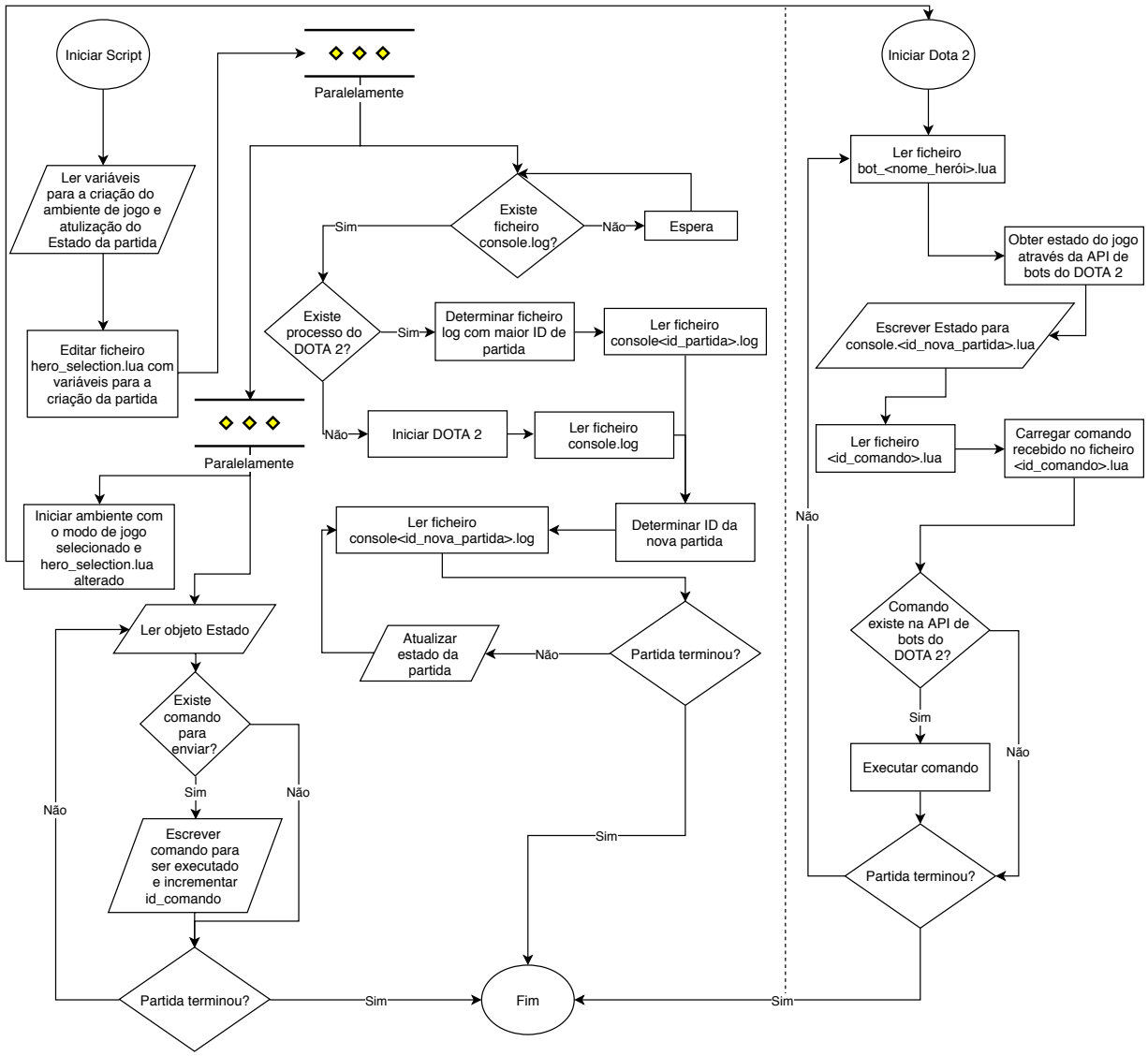


Figura 3.2: Diagrama de Fluxo de Informação do Sistema.

Algoritmo 1: Algoritmo representativo do Processo 1.

```
Result: Imprimir o estado de uma partida e executar ações representativas de
        comandos obtidos.
partida_nao_terminou = True;
while partida_nao_terminou do
    Estado = atualizar_Estado();
    print Estado;
    cmd = obter_comandos();
    if cmd == ação_existente_na_API then
        executar_ação_respetiva();
    else
    end
    if partida_terminou() then
        partida_nao_terminou = False;
    else
    end
end
```

Algoritmo 2: Algoritmo representativo do Processo 2.

```
Result: Ler o estado de uma partida e enviar comandos representativos de ações
        para o herói executar.
id_comando = 0;
partida_nao_terminou = True;
comando_para_enviar = null;
while partida_nao_terminou do
    ler_Estado();
    comando_para_enviar = obter_comando();
    if comando_para_enviar <> null then
        enviar_comando(comando_para_enviar, id_comando);
        id_comando += 1;
    else
    end
    if partida_terminou() then
        partida_nao_terminou = False;
    else
    end
end
```

Paralelamente, o *Processo 2* (Algoritmo 2) trata de ler estado de uma partida e averiguar se existe um novo comando para executar. Se existir, o comando é escrito para o ficheiro <`id_comando`>`.lua`, utilizando a variável `id_comando` que é, depois, incrementada em uma unidade. Se não existir, prossegue o processamento. Ainda neste objetivo, verifica-se se a partida já terminou. Se não terminou, este objetivo volta a repetir o seu ciclo, se terminou, o ciclo geral do sistema termina.

Algoritmo 3: Algoritmo representativo do Processo 3.

```
Result: Atualizar o estado de uma partida, através da leitura de logs de partidas.
id_partida = 0;
id_nova_partida = 0;
Estado = []
while not existe("console.log") do
    wait();
end
if existe(dota2_pid) then
    id_partida = determinar_logfile_com_maior_id();
    ler("console."+id_partida+".log");
    id_nova_partida = determinar_id_nova_partida();
else
    iniciar_DOTA2();
    ler("console.log");
    id_nova_partida = determinar_id_nova_partida();
end
partida_nao_terminou = True;
while partida_nao_terminou do
    atualizar_Estado("console."+id_nova_partida+".log", Estado);
    if partida_terminou() then
        partida_nao_terminou = False;
    else
    end
end
```

O *Processo 3* (Algoritmo 3) trata de ler e obter informação sobre o estado da partida, através de ficheiros de *log*. É feita uma verificação sobre a existência do ficheiro `console.log` que é criado pelo jogo, quando este é inicializado pela primeira vez. Enquanto o ficheiro não existir, o sistema espera; quando for criado o ficheiro, é feita uma nova verificação. Desta vez verifica-se se existe o processo do DOTA 2 no SO. Se sim, é determinado o *logfile* com maior ID de partida e é lido esse mesmo ficheiro até ser possível determinar o ID da nova partida criada. Se não existir o processo do DOTA 2, é iniciado o DOTA 2 e lido o ficheiro `console.log`, de modo a determinar o ID da nova partida criada. Estando o ID da nova partida criada determinado, quer seja através do

ficheiro `console.log` ou do ficheiro `console.<id_partida>.log`, é finalmente lido o ficheiro `console<id_nova_partida>.log`, de modo a obter um novo estado da partida. Enquanto a partida não terminar, objeto Estado é atualizado, voltando-se a ler o ficheiro, repetindo o ciclo. Quando a partida terminar, o ciclo geral do sistema termina.

3.3 O *Script* LUA do Herói

De maneira a controlar um herói através de um agente em Python, com esta *framework* é necessário criar um *script* LUA para o herói respetivo, recorrendo à API oficial de *scripting* de *bots* de DOTA 2. Para efeito de prova de conceito denvolveu-se um *script* para o herói *Ursa*. O *script* criado em LUA para este sistema, divide-se em duas partes fundamentais:

- A **obtenção do estado do jogo** através da API de *bots*. O estado do jogo é composto pelo **estado das características básicas do herói**, bem como pelo **estado do Mundo**, constituído pelo **estado das características das unidades aliadas**, **adversárias**, **neutras** e do **mapa**;
- A **receção de comandos** enviados através do módulo desenvolvido (`PythonDota2`) e a **execução** dos mesmos, através da API de *bots*.

A obtenção do estado do herói é efetuada, usando as funções:

- `GetBot()`, **função globa**l, que devolve um *handle* para o herói, denominado `bot`. Todas as outras funções que retornam o estado de alguma característica do herói utilizam este *handle*;
- `bot:GetLocation()`, **função do âmbito desta unidade** (`bot`), que devolve a localização do herói, sob a estrutura de dados `Vector` de coordenadas x, y e z;
- `bot:GetMana()`, função que devolve a *mana* do herói, sob o tipo de dados *int*;
- `bot:GetMaxMana()`, função que devolve a *mana* máxima do herói, sob o tipo de dados *int*;
- `bot:GetHealth()`, função que devolve os pontos de vida do herói, sob o tipo de dados *int*;
- `bot:GetMaxHealth()` função que devolve os pontos de vida (HP) máximos do herói, sob o tipo de dados *int*;
- `bot:NumQueuedActions()`, função que devolve o número de ações presentes na fila, à espera de serem executadas, sob o tipo de dados *int*;

De modo a obter o estado do Mundo, foram utilizadas as funções:

- `bot:GetNearbyNeutralCreeps(nRadius)`, função do âmbito do herói (`bot`), que devolve *handles* para os *creeps* neutros localizados num raio de *nRadius* unidades de distância do herói, sob a forma de uma Tabela, cujos elementos estão ordenados por ordem crescente de distância ao herói;

- `bot:GetNearbyLaneCreeps(nRadius, bEnemies)`, função que devolve *handles* para os *creeps* adversários localizados num raio de *nRadius* unidades de distância do herói, sob a forma de uma Tabela, cujos elementos estão ordenados por ordem crescente de distância ao herói;

- `GetUnitList(UNIT_LIST_ENEMY_HEROES)`, função global que devolve *handles* para os heróis adversários, em qualquer ponto do mapa, sob a forma de uma Tabela;

- `GetUnitList(UNIT_LIST_ENEMY_BUILDINGS)`, função global que devolve *handles* para os edifícios adversários, em qualquer ponto do mapa, sob a forma de uma Tabela;

- `DotaTime()`, função global, que devolve o tempo decorrido da partida, sob o tipo de dados *float*;

- `GetTimeOfDay()`, função global, que devolve a altura do dia (dia ou noite), sob o tipo de dados *float*;

- `GetGameState()`, função global, que devolve o estado em que a partida se encontra, sob o tipo de dados *int*.

As quatro primeiras funções que permitem obter o estado do Mundo, listadas anteriormente, apenas devolvem Tabelas de *handles* para unidades do jogo, pelo que é necessário iterar as Tabelas e obter informações sobre os elementos das mesmas, através das seguintes funções:

- `unit:GetUnitName()`, função do âmbito da unidade (`unit`), que devolve o nome da mesma, sob o tipo de dados *string*;

- `unit:GetHealth()`, função que devolve os pontos de vida (HP) da unidade, sob o tipo de dados *int*;

- `unit:GetMaxHealth()`, função que devolve os pontos de vida (HP) máximos da unidade, sob o tipo de dados *int*;

- `unit:GetMana()`, função que devolve a *mana* da unidade, sob o tipo de dados *int*;

- `unit:GetMaxMana()`, função que devolve a *mana* máxima da unidade, sob o tipo de dados *int*;

- `unit:GetAttackDamage()`, função que devolve o dano de ataque da unidade, sob o tipo de dados *float*;

- `unit:GetArmor()`, função que devolve a armadura da unidade, sob o tipo de dados *float*;
- `GetUnitToUnitDistance(unit, bot)`, função global que devolve a distância desde a unidade (`unit`) até ao herói (`bot`), sob o tipo de dados *float*;

Este é apenas um pequeno conjunto das funções que podem ser utilizadas e que é apenas demonstrativo. Existem muitas outras funções, presentes na API de *bots* de DOTA 2, que podem ser utilizadas para obter o estado do jogo, mas estas são as mais básicas e, portanto, as mais pertinentes e importantes.

O *script* LUA criado para este herói funciona como modelo base para qualquer herói que se pretenda controlar através de um agente desenvolvido em Python e encontra-se no sistema criado neste trabalho. As funções disponíveis na interface criada em Python para controlo do agente encontram-se descritas na Secção 3.4.

A função `Think()` presente nos *scripts* LUA dos *bots* é executada uma vez por cada nova ***tick*** do jogo, como refere [12]. Uma *tick* é o intervalo de tempo que o **servidor** de DOTA 2 toma para computar toda a informação relacionada com a partida, bem como as ações tomadas pelas unidades, criando um novo estado do jogo, devolvendo o mesmo ao **cliente** de DOTA 2. Este processo está ilustrado na Figura 3.3.

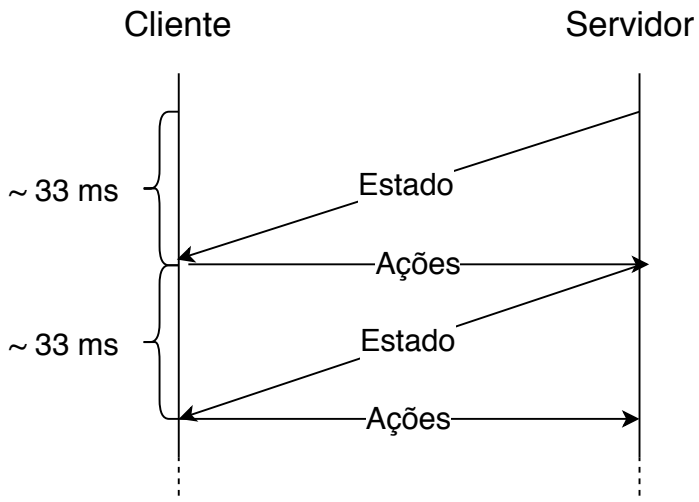


Figura 3.3: Processo de comunicação entre o Cliente e o Servidor de *ticks* de uma partida de DOTA 2.

A Figura 3.4 ilustra uma captura de ecrã retirada de uma partida de DOTA 2. Nesta partida é controlado um herói, através de um *script* LUA, que imprime o resultado da função `DotaTime()`, que retorna o tempo da partida, em segundos, sob o tipo de dados *float* [12]. É possível verificar que existe uma diferença de 0.033 segundos (33 milissegundos) entre as duas linhas selecionadas na Figura. Portanto, é possível concluir que uma *tick* tem uma duração de 0.033 segundos e que o estado do jogo é atualizado, através da API de *bots* de DOTA 2 e escrito para um *logfile*, a uma frequência de trinta *Hertz*. Assim sendo,

comprova-se que o *script* é executado a cada *tick* da partida. Imprimindo o estado do jogo apenas numa linha, que contém a informação obtida através das funções anteriormente detalhadas nesta Secção, o *logfile* correspondente à partida em questão irá conter um estado do jogo por cada *tick* que passa, ou seja, 30 estados do jogo diferentes num segundo.

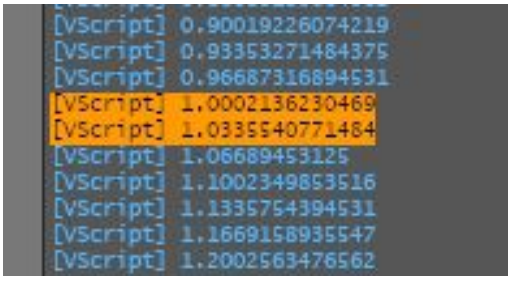


Figura 3.4: Captura de ecrã demonstrativo da frequência de *ticks* do DOTA 2. É possível verificar que a diferença de tempo nas duas linhas selecionadas é de aproximadamente 0.033 segundos, demonstrando que cada *tick* do jogo tem aproximadamente essa duração.

Relativamente à execução de comandos no *script* do *bot* a controlar, esta começa na receção de comandos, enviados através do módulo desenvolvido (`PythonDota2`). Estes comandos são escritos para ficheiros LUA (apenas um comando por ficheiro), que ficarão guardados numa pasta temporária (`tmp`). O conteúdo destes ficheiros é um simples bloco de código em LUA que retorna uma *string*, que contém o nome de uma função presente na API de *bots* de DOTA 2, podendo conter também argumentos. É utilizada uma função de LUA (`loadfile` [47]), que carrega o bloco de código presente nos ficheiros LUA, sem o executar. É bastante útil usar este mecanismo porque a função `loadfile` devolve `nil` caso haja algum erro carregar o bloco de código, ou seja, o erro possível é não existir um ficheiro que contenha o comando enviado, o que significa que não foi enviado um comando. Caso a função `loadfile` não devolva `nil`, esta pode ser finalmente, executada. Por fim, a execução da função carregada devolve uma *string*. Se esta *string* começar pelo nome de alguma função presente na API de *bots*, a *string* é dividida em partes, por espaços, e a função da API correspondente, será executada, com os argumentos (partes restantes divididas da *string*) respetivos, se existirem. Por exemplo, imagine-se que existe um ficheiro LUA com o conteúdo "`return ''Action_MoveDirectly 0 0 0;''`". Este ficheiro será carregado como uma função de LUA, a partir do *script* do herói que se pretende controlar, utilizando a função `loadfile`, verificando-se se não é devolvido `nil`. Uma vez feita esta verificação, é executada a função carregada, que devolve uma *string*, verificando-se se esta começa pelo mesmo nome de alguma das funções presentes na API de *bots*. Se isto se verificar, a *string* devolvida pela função executada é dividida e a segunda, terceira e quarta partes da *string* ("0", "0" e "0") são transformadas para número utilizando a função de LUA `tonumber` [48], inseridas num `Vector`, que é, finalmente, utilizado como argumento da função `Action_MoveDirectly( vLocation )`. Esta função é executada e o herói controlado começa a movimentar-se para a posição (0,0,0), que se encontra, sensivelmente, a meio do mapa de DOTA 2.

3.4 O Módulo PythonDota2

Este módulo foi desenvolvido, numa primeira fase, recorrendo ao módulo `threading` [49] e, numa fase final, com recurso ao módulo `multiprocessing` [50]. Ambos os módulos fazem parte da biblioteca padrão do Python, pelo que não existe necessidade de instalar nenhum deles no SO. Houve necessidade de implementação de concorrência devido à necessidade de obter estados de partidas continuamente, sem que fosse afetado o processo de aprendizagem e teste dos agentes desenvolvidos em Python. Assim, a obtenção de estados apenas seria possível se efetuada paralelamente à execução do processo já referido.

O módulo `PythonDota2` possui uma classe homónima, na implementação com o módulo `threading` ("implementação *threaded*"), que é composta por vinte funções. Seis destas funções são privadas, para que o utilizador do módulo não lhes possa aceder e as restantes catorze são públicas.

Na implementação com o módulo `multiprocessing` ("implementação *multiprocessed*") existem trinta e oito funções, na classe `PythonDota2`, tendo vinte funções o mesmo comportamento, ou um comportamento similar às funções da implementação *threaded*. As restantes dezoito funções foram adicionas a esta implementação de modo a complementar a mesma. Dezassete destas funções são apenas necessárias para retornar atributos do estado do jogo, sendo uma destas funções, privada; na implementação *threaded* não são necessárias funções para retornar os valores dos atributos do estado do jogo, basta usá-los diretamente nos *scripts* criados que utilizam o módulo `PythonDota2`. A função restante que foi implementada apenas serve para complementar a automatização através de GUI, mais concretamente, para terminar o ambiente da partida de DOTA 2. Nesta implementação, *multiprocessed*, foi necessário criar uma classe denominada `State`, que é composta por trinta e seis funções, que são *getters* e *setters* dos atributos do estado do jogo. A classe `State` tem a finalidade de guardar estes atributos, para que sejam atualizados, pois existem diferenças entre o processamento paralelo usando o módulo *threading* e o processamento paralelo usando o módulo *multiprocessing*, nomeadamente na partilha de variáveis entre funções de uma classe. Estas diferenças levaram ao desenvolvimento da classe `State` na implementação *multiprocessed* e não na implementação *threaded*. A classe State possui, assim, os atributos do estado do jogo, anteriormente presentes na implementação *threaded* na classe `PythonDota2`.

Seguidamente, na Subsecção 3.4.1, encontram-se descritas as funções representativas de ações que o herói pode tomar, bem como a implementação geral do sistema, que é comum na implementação *threaded* e na implementação *multiprocessed*, bem como as componentes das mesmas (Subsecção 3.4.2). As diferenças entre as duas implementações serão explicadas na Subsecção 3.4.3 desta Secção.

3.4.1 Funções Disponibilizadas pelo Módulo Desenvolvido

Seguidamente encontram-se descritas as 10 funções representativas de ações que um herói de DOTA 2 controlado por um agente desenvolvido em Python pode tomar, existentes no módulo `PythonDota2`, replicando as funções presentes na API oficial de *scripting* de *bots*

de DOTA 2 [12].

- `Action_MoveDirectly(x, y)`, que comanda o herói a deslocar-se até à localização dada pelas coordenadas x e y. Esta função replica a função `Action_MoveDirectly(vLocation)` da API de *scripting* de *bots*;

- `ActionPush_MoveDirectly(x, y)`, que executa um *push* para o início da fila duplamente terminada a ação de o herói se deslocar até à localização dada pelas coordenadas x e y. Esta função replica a função `ActionPush_MoveToLocation(vLocation)` da API de *scripting* de *bots*;

- `ActionQueue_MoveDirectly(x, y)`, que executa uma *queue* para o fim da fila duplamente terminada a ação de o herói se deslocar até à localização dada pelas coordenadas x e y. Esta função replica a função `ActionQueue_MoveToLocation(vLocation)` da API de *scripting* de *bots*;

- `Action_AttackUnit(handle, once)`, que comanda o herói a atacar a unidade determinada pelo *handle*, uma vez se *once* for `True` ou indefinidamente se *once* for `False`. Esta função replica a função `Action_AttackUnit(hUnit, bOnce)` da API de *scripting* de *bots*;

- `ActionPush_AttackUnit(handle, once)`, que executa um *push* para o início da fila duplamente terminada a ação de o herói atacar a unidade determinada pelo `handle`, uma vez se *once* for `True` ou indefinidamente se *once* for `False`. Esta função replica a função `ActionPush_AttackUnit(hUnit, bOnce)` da API de *scripting* de *bots*;

- `ActionQueue_AttackUnit(handle, once)`, que executa uma *queue* para o fim da fila duplamente terminada a ação de o herói atacar a unidade determinada pelo `handle`, uma vez se *once* for `True` ou indefinidamente se *once* for `False`. Esta função replica a função `ActionQueue_AttackUnit(hUnit, bOnce)` da API de *scripting* de *bots*;

- `Action_ClearActions(stop)`, que limpa a fila duplamente terminada de ações, sendo possível parar qualquer ação que o herói esteja a executar, se *stop* for `True`. Esta função replica a função `Action_ClearActions(bStop)` da API de *scripting* de *bots*;

- `Action_Delay(delay)`, que comanda o herói a esperar determinado tempo, dado pelo argumento *delay*, em segundos. Esta função replica a função `Action_Delay(fDelay)` da API de *scripting* de *bots*;

- `ActionPush_Delay(delay)`, que executa um *push* para o início da fila duplamente terminada a ação de o herói esperar determinado tempo, dado pelo argumento *delay*, em segundos. Esta função replica a função `ActionPush_Delay(fDelay)` da API de *scripting* de *bots*;

- `ActionQueue_Delay(delay)`, que executa uma *queue* para o fim da fila duplamente terminada a ação de o herói esperar determinado tempo, dado pelo argumento *delay*, em segundos. Esta função replica a função `ActionQueue_Delay(fDelay)` da API de *scripting* de *bots*.

Existem muitas outras funções representativas de ações que heróis possam tomar que poderiam ser implementadas mas, para a demonstração de utilização do sistema desenvolvido, para o desenvolvimento de agentes de DOTA 2 em Python, as funções descritas anteriormente são suficientes.

3.4.2 Implementações do Módulo

No módulo `PythonDota2`, a sua classe homónima, possui uma função de inicialização (`__init__()`) onde são inicializados alguns atributos de classe, nomeadamente localizações de ficheiros importantes, uma variável de condição [51, 52] para estabelecer sincronismo entre o envio de comandos e a leitura do estado do jogo e variáveis que serão úteis para iniciar uma partida através da GUI do DOTA 2. O conjunto de variáveis representativas do estado do jogo também se encontra nesta função, na implementação *threaded*. Na implementação *multiprocessed*, estas variáveis encontram-se na classe `State`. Ainda nesta função, é chamada a função (`__edit_hero_selection(hero_to_pick, enemy_to_pick)`) que trata da edição do ficheiro `hero_selection.lua` com base nos argumentos da função, bem como na variável `team` da classe `PythonDota2`, que representa a equipa do herói a controlar. Para além da edição do ficheiro, é iniciada, paralelamente, a função responsável pela leitura da atualização do estado do jogo (`read_log()`). Na implementação *threaded* esta função é iniciada com o uso de uma *Thread* e na implementação *multiprocessed* a função é iniciada com o uso de um *Process* (processo).

A função que trata da edição do ficheiro `hero_selection.lua` guarda, em memória, o conteúdo do mesmo e altera esse conteúdo guardado em memória, com base na equipa e no herói que se pretende controlar, bem como no herói adversário pretendido. Por fim é escrito o conteúdo alterado, guardado em memória, para o ficheiro de seleção dos heróis.

De maneira a obter o estado das partidas, a leitura de *logfiles* é executada através do comando *tail* [53], presente em SOs da família Unix. Este comando é executado através da função `read_log()`, que é executada em paralelo com o processo principal. É usada a opção "-f" no comando *tail*, de modo a obter novas linhas que sejam adicionadas ao fim do ficheiro, juntamente com uma técnica de *polling*, que consiste em verificar repetidas vezes o estado do ficheiro; se o estado não for alterado (se não existir uma nova linha), o sistema espera; se o estado for alterado (se existir uma nova linha) o sistema é avisado e a nova linha é recuperada. Esta técnica facilita a leitura do *logfile*, pois não é necessário esperar tempo explicitamente definido para voltar a verificar o fim do ficheiro, o que pode introduzir tempos de espera indesejáveis ao sistema.

Entretanto é iniciada a nova partida, através da função `start_env_lobby()`, que usa um módulo denominado `pyautogui`, utilizado para controlar o rato e o teclado, programaticamente [54] e um módulo denominado `xdotool` [55], utilizado para alterar o foco

de janelas, num ambiente gráfico em Linux. Este módulo foi utilizado para efetuar uma automatização através da GUI do DOTA 2, de forma a iniciar uma partida num *lobby* local. Apesar de este ser local, é necessária uma conexão à Internet pois é feita a criação de uma partida de DOTA 2 com um ID de partida, que tem de ser único. A automatização funciona com base em reconhecimento de imagens presentes no ecrã, cliques do rato em pontos específicos ou relativos a imagens reconhecidas e, também, escrita de texto na consola do DOTA 2, para executar alguns comandos que fazem parte da automatização. Assim, com base na equipa, herói a controlar, herói adversário, modo de jogo e número de episódios a executar escolhidos, quando instanciada a classe `PythonDota2`, é criada a partida e repetida se o número de episódios for maior do que 1. A função `close_env_lobby()` é semelhante à função referida anteriormente mas, em vez de iniciar o ambiente de uma partida de DOTA 2, termina-o, recorrendo a automatização através da GUI do jogo.

A função de criação de partida, pode ser antecipada pela função `start_dota()`, que executa um comando, através do módulo `subprocess` [56], que inicia o DOTA 2 através do Steam [46], se não existir nenhum processo do jogo, no SO. Se existir um processo, o foco do SO muda para o jogo. Se o Steam não tiver processo ativo no SO, será iniciado, bem como o DOTA 2, de seguida.

Uma vez criada a nova partida, é então verificado no *logfile*, através da função que corre paralelamente (`read_log()`), se foi criada uma nova partida; até existir a informação de que foi criada a partida, o *logfile* continua a ser lido; quando for obtida a informação de que a nova partida foi criada, com um ID específico (`id_nova_partida`), é concluída a leitura do *logfile* e iniciada a leitura do ficheiro de *log* da nova partida, denominado `console.<id_nova_partida>.log`. A partir deste ficheiro, é finalmente lido o estado da partida, que se iniciou através do processo principal, com automatização através da GUI do jogo. O estado da partida, como referido na Secção 3.3 deste Capítulo, é atualizado 30 vezes num segundo, pois cada *tick* do jogo tem a duração de 0.033 segundos, pelo que o *logfile* tem de ser lido a uma velocidade compatível, de modo a não existir falha de leitura de alguma *tick*. O estado do jogo, impresso no *logfile*, é constituído apenas por valores, separados, entre si, pelo ";". A ordem com que os valores são impressos é a mesma com que é feito o *parsing* do *logfile*, para que o estado obtido faça sentido. Depois do processamento do estado do jogo (contido numa linha), cada variável é guardada para o módulo `PythonDota2`, para poderem ser utilizadas, por exemplo, num algoritmo de ML. Esta operação (leitura e atualização do estado do jogo no módulo) é efetuada recorrendo a um *mutex* presente numa variável de condição [51, 52], para garantir, através da função `acquire()`, que o acesso aos recursos (todas as variáveis do módulo, referentes ao estado do jogo) é feito em exclusão mútua. Uma fez finalizada a operação, são utilizadas as funções `notify_all()` e `release()` para notificar outras *threads* ou processos de que o acesso às variáveis terminou e para libertar o acesso aos recursos, podendo o acesso ser tomado por outra *thread* ou processo.

O envio de comandos de ações para o *bot*, através deste módulo, é efetuado usando uma função (`send_command(command)`) que escreve, para um ficheiro LUA, uma função que retorna uma *string*, representativa do comando (`command`) que se pretende enviar. O ficheiro LUA, finalmente escrito, é interpretado pelo *script* do *bot* e é executada a ação

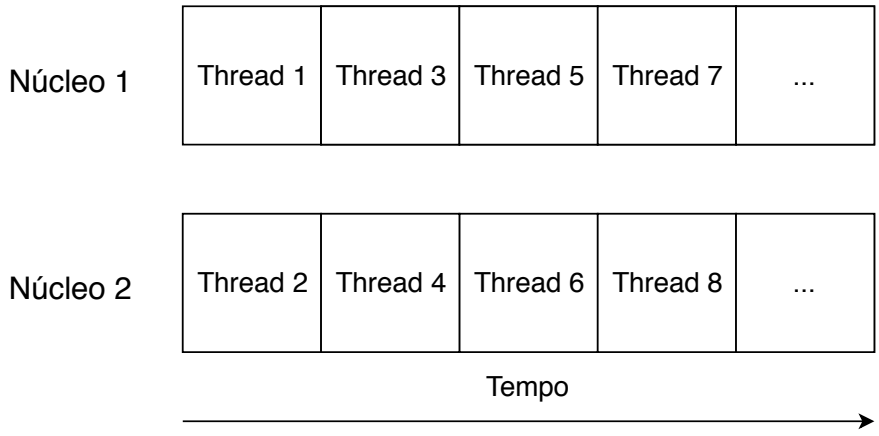


Figura 3.5: Processador, constituído por dois núcleos, que paraleliza pares de *threads*. *Thread* 1 paralela a *thread* 2, *thread* 3 paralela a *thread* 4 e assim sucessivamente.

respetiva à *string* obtida através do ficheiro, como já referido na Secção 3.3 deste Capítulo.

A função `send_command(command)` é chamada através de funções, presentes no módulo `PythonDota2`, que representem uma ação que exista na API de *bots* de DOTA 2 [12]. Por exemplo, se se pretendesse controlar o deslocamento do *bot* até uma dada localização, poder-se-ia utilizar uma das funções da API de *bots* que executasse movimentação do herói. Uma destas funções é a `Action_MoveDirectly(vLocation)`, cujo argumento é um `Vector` de coordenadas x, y e z. Utilizando a API desenvolvida, é possível usar a função `Action_MoveDirectly(x, y)`, cujos argumentos são duas coordenadas, suficientes para determinar a localização pretendida, pois a coordenada z é irrelevante no tipo de comandos de movimentação do herói. Esta função forma um comando, sob o tipo de dados *string*, que é enviado, como argumento, através da função `send_command(command)`. O *script* do herói a controlar irá obter o comando através de um ficheiro LUA e executará a ação pretendida.

3.4.3 Diferenças entre Implementações de Paralelismo do Sistema

Como referido na Subsecção 3.4.2, a função `read_log()`, do módulo `PythonDota2`, é executada paralelamente ao processo principal, criado através da utilização deste módulo, quando instanciada a classe `PythonDota2`. Este paralelismo foi, inicialmente, implementado com recurso ao módulo de *threading* [49], de Python [57] e, mais tarde, com recurso ao módulo de *multiprocessing* [50], da mesma linguagem de programação.

Um processador, constituído por um único núcleo, tem apenas a capacidade de executar múltiplas *threads* consecutivamente, pois apenas é executada uma *thread* de cada vez. Já um processador, constítuido por múltiplos núcleos, tem a capacidade de paralelizar tarefas (*threads*), pois em diferentes núcleos podem ser executadas diferentes *threads*, uma por núcleo [58]. A Figura 3.5 ilustra o modo de paralelização referido.

No entanto, tal mecanismo não se verifica em Python. Isto está diretamente relacio-

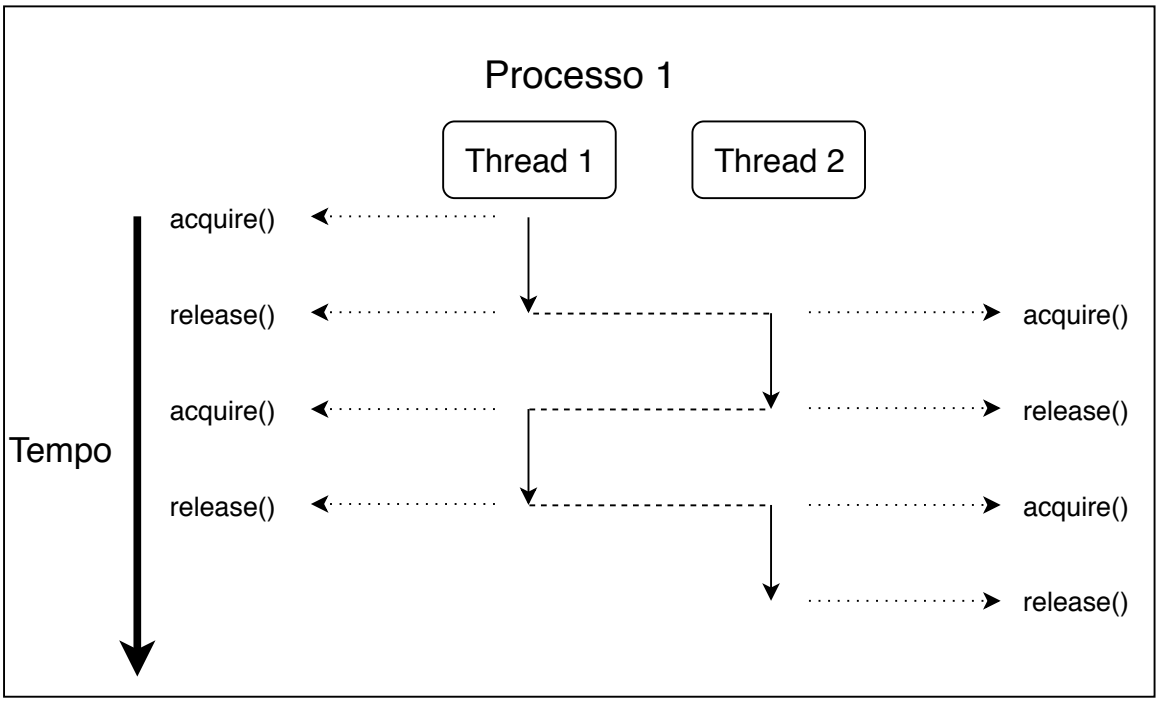


Figura 3.6: Funcionamento de duas *threads*, em Python, no mesmo processo. Nesta linguagem de programação as *threads* não são executadas paralelamente, mas sim concorrentemente, devido ao GIL [60].

---

nado com CPython [59], que é a implementação original da linguagem Python. CPython, desenvolvida com a utilização das linguagens de programação C e Python, possui um GIL (*Global Interpreter Lock*) [60], que é um *mutex* que restringe o acesso a objetos Python apenas à *thread* que o usa, em determinado instante [61]. Este mecanismo acrescenta segurança ao acesso partilhado de objetos Python, mas retira o paralelismo que é subentendido com o uso de *threads*. Não é, portanto, possível, num programa Python constituído por um processo, contendo pelo menos, duas *threads*, que estas sejam executadas paralelamente. Assim sendo, um sistema composto por mais do que um núcleo não tira proveito do uso de *threads*, utilizando Python como linguagem de programação. A Figura 3.6 ilustra o funcionamento de duas *threads* num processo em Python, tal como o funcionamento do GIL.

Apesar de não ser possível paralelizar *threads* num mesmo processo, em Python, a partilha de objetos, entre *threads*, é menos complexa do que a partilha de objetos entre processos. Múltiplas *threads*, constituintes de um processo, podem partilhar determinados recursos do mesmo. Vários processos não partilham memória entre si, pelo que é necessária a implementação de estruturas partilhadas entre processos [58]. A Figura 3.7, adaptada de [58] retrata as diferenças entre um processo composto por uma *thread* e outro processo composto por múltiplas *threads*. Analisando a Figura 3.7, é possível concluir que todas as *threads* constituintes de um processo podem ter acesso ao código, informação e ficheiros desse mesmo processo, possuindo, cada *thread*, os seus registos e pilha.

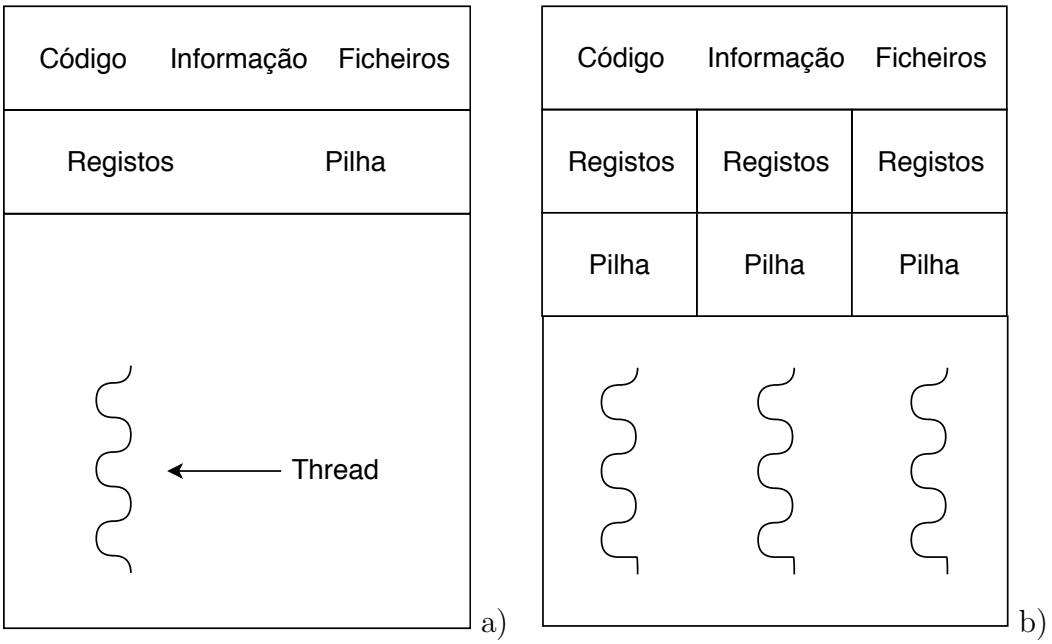


Figura 3.7: Comparação entre um processo composto por uma *thread* e um processo composto por múltiplas *threads*. a) Processo composto por apenas uma *thread*. b) Processo composto por múltiplas *threads* [58].

Em Python, é possível implementar partilha de recursos entre processos de duas formas distintas, através de:

- *Shared Memory* (Memória Partilhada) [62]. Este modelo envolve a criação de uma região de memória, sobre a qual vários processos possam escrever e ler informação, partilhando, assim, recursos. A Figura 3.8 ilustra este mecanismo;

- *Server Process Manager* (Gestor de Processo em Servidor) [62]. Esta forma de partilha de recursos entre processos tem o objetivo de gerir um processo possuidor de objetos Python, que forneça a manipulação, a outros processos, dos objetos a partilhar. A Figura 3.9 ilustra este mecanismo.

Para a implementação de paralelismo entre processos, no módulo `PythonDota2` foi utilizado um *Server Process Manager* customizado. Foi feita esta escolha pois, apesar de não ser tão eficiente como *Shared Memory* [62], um *manager* permite a partilha de estruturas de dados Python como, por exemplo, `list` [63] ou `dict` [64], contrariamente à *Shared Memory*.

O *manager* customizado foi criado, tendo, como superclasse, a classe `BaseManager` [65], registando uma nova classe (`State`), para ser gerida pelo *manager*. Este registo da classe é efetuado utilizando a função `register(typeid, callable)` [66], que fornece uma classe `State` *callable* para os processos. Esta classe tem o objetivo de guardar o estado do jogo de DOTA 2 em variáveis, bem como disponibilizar funções, *getters* e *setters*, que permitam

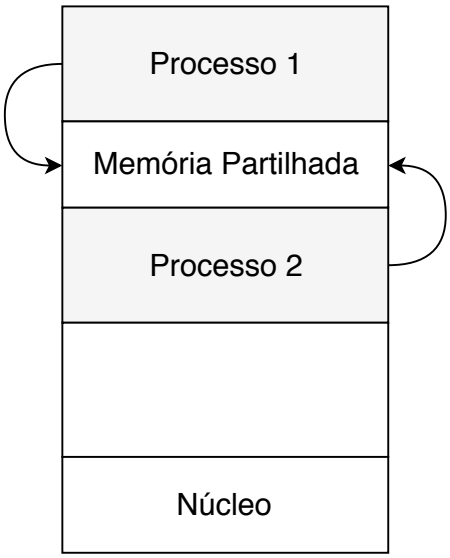


Figura 3.8: Memória Partilhada entre dois processos presentes no mesmo processador, adaptado de [58].

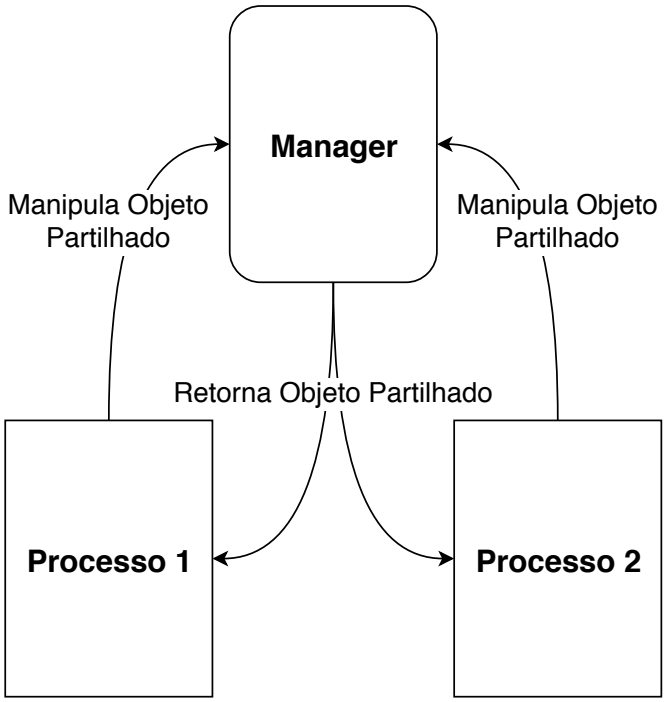


Figura 3.9: Partilha de recursos através de um *manager* que retorna objetos Python de modo a serem partilhados entre múltiplos processos.

a leitura e reescrita dessas mesmas variáveis, respetivamente, através da classe principal, `PythonDota2`. Assim, as variáveis do estado do jogo podem ser partilhadas entre processos, mantendo o paralelismo, entre os mesmos, intacto.

Resumidamente, a grande diferença entre as duas implementações é o uso de processos em vez das iniciais *threads*. O uso de processos permite execução paralela de tarefas, contrariamente às *threads*, em Python. Devido a ter sido reimplementado o módulo `PythonDota2` com recurso a processos, foi necessário mudar um pouco a implementação, especialmente na partilha de objetos Python entre processos e classes.

Na primeira implementação deste módulo (com *threads*), o estado do jogo de DOTA 2 podia ser guardado em atributos da classe `PythonDota2`, e a sua alteração podia ser feita apenas alterando o valor dos respetivos atributos, através de atribuição de novos valores aos atributos. O utilizador podia, assim, usar os atributos diretamente no seu código, uma vez instanciada a classe `PythonDota2`.

Com a implementação efetuada com recurso a processos, o estado do jogo é guardado em atributos de uma nova classe, chamada `State`, e esses atributos são lidos e alterados através da classe principal (`PythonDota2`), através de funções *getters* e *setters*, fornecidas pela classe `State`, registada num *manager* de processos customizado, que permite a partilha da informação do estado do jogo entre vários processos. Assim, com esta implementação, o utilizador, em vez de utilizar atributos da classe `PythonDota2` diretamente, pode utilizar funções desta classe, que devolvem valores guardados nos atributos da classe `State`. Como já observado anteriormente, a implementação com uso de processos permite computação paralela, contrariamente à implementação com recurso a *threads*. Por estes motivos escolheu-se, para a versão final desta *framework*, utilizar a implementação com processos.

3.5 Conclusão

Neste Capítulo foi abordado o tema da arquitetura e implementação da *framework* desenvolvida, em detalhe, começando por descrever as características e detalhes da arquitetura do sistema desenvolvido e do fluxo de informação no sistema, passando por uma explicação do método de implementação e funcionamento do *script* LUA, que é responsável por obter o estado das partidas, imprimi-lo para *logfiles* e receber comandos representativos de ações que um agente de DOTA 2 possa tomar. De igual forma, foi detalhado o processo de desenvolvimento do módulo `PythonDota2`, com uma descrição das funções disponíveis no mesmo, representativas de ações que os heróis de DOTA 2 possam tomar, tal como o processo de decisão de implementação de certos aspetos, nomeadamente a escolha entre um processo ou uma *thread* para efetuar paralelização da função de leitura de *logfiles* de partidas de DOTA 2, com o processo principal de controlo do agente. Outro aspeto importante da implementação do módulo, relativo à escolha da partilha de recursos entre processos, em Python, encontra-se discriminado no final deste Capítulo.

Capítulo 4

Testes e Resultados

Este Capítulo tem como objetivo a demonstração de testes efetuados ao sistema criado, de modo a avaliar se o mesmo é eficiente e se, para além disso, pode ser realmente utilizável para o desenvolvimento de agentes controlados por *software*, para DOTA 2. Foram efetuados 4 tipos de testes, nomeadamente:

1. Um teste de comparação de desempenho de leitura e *parsing* do estado de partidas, contido em ficheiros *log*, entre as duas implementações elaboradas (com uso de *threads* [49] e com uso de processos [50]), do módulo `PythonDota2`, como descritas no Capítulo 3 desta dissertação (Secção 4.1);

2. A demonstração de utilização do sistema desenvolvido para a implementação de um agente em Python. Esta demonstração baseia-se uma política *hard-coded*, que recorre ao envio de comandos para o herói a controlar, bem como da obtenção de estados de partidas. Esta política retrata a mecânica de jogo chamada *stacking*, que consiste em aumentar o número de *creeps* neutros de um campo na *jungle* do mapa de DOTA 2, neste caso na parte do mapa relativa à equipa *Dire*. É também testado o envio de comandos para o agente, de forma a averiguar se vários comandos seguidos são, realmente, enviados consecutivamente para o DOTA 2, através do módulo `PythonDota2`, e em que *tick*, no *logfile* da respetiva partida, os comandos são executados (Secção 4.2);

3. Um teste de consistência de dados, que consiste em verificar se, através do módulo `PythonDota2`, as variáveis do estado de partidas têm exatamente os mesmos valores presentes nos *logfiles* das respetivas partidas (Secção 4.3);

4. Um teste comparativo entre o acesso a variáveis do estado de partidas, através do módulo `PythonDota2`, com recurso a uma função chamada `wait_for_tick()` e sem o recurso a esta função, em que se testa, também, em cada caso, o desempenho do acesso às variáveis (Secção 4.4).

Todos os testes foram efetuados com recurso à implementação do módulo `PythonDota2` feita com processos. O teste 1, para além desta implementação, utiliza também a implementação feita com *threads*. Os testes 3 e 4 são efetuados com base na política testada

no teste 2, sendo, esta política, mais pertinente no próprio teste 2, pois testa o envio de comandos, a partir do módulo `PythonDota2`, bem como a receção dos mesmos no *script* do herói respetivo ao agente de DOTA 2 que se pretende controlar.

Em todas as execuções de testes de partidas foi aumentada a escala de tempo ao máximo, com recurso à automatização através de GUI, utilizando, no terminal do DOTA 2, o comando `host_timescale 100` [67] (Figura A.6). O valor 100 não é o valor máximo deste comando, nem aumenta a escala de tempo das partidas em 100 vezes, apenas foi utilizado este valor para se ter a certeza que o aumento da escala de tempo era o maior possível. Na documentação sobre os comandos existentes para o terminal de DOTA 2 [67] não é indicado o valor máximo deste comando, mas segundo os valores obtidos nos testes, o valor máximo teorizado será de 10, nunca se tendo, no entanto, obtido um *Speedup* (Equação 4.1) igual ou superior a 10, nos testes efetuados. Com o aumento da escala de tempo das partidas, o *Speedup* (Equação 4.1) de processamento das mesmas aumenta também, levando a um aumento da taxa de atualização dos estados das partidas de 30 Hz (escala de tempo normal) até ao valor teórico máximo de 300 Hz (se o *Speedup* máximo obtido for de 10). O aumento da escala de tempo da partida tem apenas a finalidade de acelerar o processo de testes e pode ser muito útil para acelerar o processo de aprendizagem e teste dos agentes desenvolvidos em Python. Mesmo acelerando ao máximo as partidas, nos testes efetuados não ocorreram falhas de obtenção de estados a partir de *logfiles*.

Com o aumento da escala de tempo das partidas, os tempos de execução de cada teste de cada partida são sempre menores que o tempo real da partida. O *Speedup* (aceleração) do tempo de execução do teste em relação ao tempo real da partida é calculado de acordo com a seguinte equação:

$$Speedup = T_p/T_e \tag{4.1}$$

em que T_p é o tempo da partida e T_e é o tempo de execução do teste.

Estes testes foram executados todos na mesma máquina, sempre sob as mesmas condições, de modo a que as comparações entre determinados testes sejam justas. A máquina onde foram executados os testes tem as seguintes características:

- SO Ubuntu 18.04.1 LTS;
- Uma CPU Intel Core i7-5500U, com uma frequência de relógio de 2.40 GHz;
- 8 GB de RAM (*Random-Access Memory*);
- Uma GPU GeForce 920M/PCIe/SSE2;
- Uma HDD (*Hard Disk Drive*).

4.1 Desempenho de Leitura de Ficheiros de *log* com Processo e *Thread*

Foram feitos dois testes para analisar o desempenho da leitura, *parsing* e atribuição de valores obtidos a objetos de classes, de *logfiles*, utilizando a implementação com *threads* e

a implementação com processos, da função `read_log()`.

O primeiro teste consistiu em criar uma partida de DOTA 2, cujo *logfile* fosse lido até serem atingidos 300 segundos de partida, sem contar com os 90 segundos da fase de preparação, perfazendo um total de 390 segundos totais de partida. Com a utilização de uma *thread* para executar a função `read_log()`, responsável por ler, efetuar *parsing* e atribuir os valores obtidos a objetos da classe `PythonDota2`, o número de ciclos de processador efetuados foi de 21009, com uma duração de, aproximadamente, 53.402 segundos, tendo-se obtido uma média de 0.0025 segundos por ciclo. A partida (390 segundos) foi, portanto, acelerada em, aproximadamente, 7.303 vezes, de acordo com a equação 4.1. De igual modo, foi criada uma outra partida com as mesmas condições, mas, com a leitura e *parsing* do *logfile* a serem efetuados por um processo, atribuindo os valores obtidos a objetos da classe `State`. Esta execução teve um total de 40246 ciclos de processador, com uma duração total de 52.272 segundos, tendo-se obtido uma média de 0.0013 segundos por ciclo. Esta nova partida foi acelerada em, aproximadamente, 7.461 vezes.

O segundo teste consistiu em criar uma partida de DOTA 2, cujo *logfile* fosse lido até serem atingidos 10000 ciclos, através de uma *thread* e de um processo. O uso de *thread* teve uma duração aproximada de 18.683 segundos a executar 10000 ciclos da função *read_log()*, tendo, portanto uma duração aproximada de 0.0019 segundos por ciclo, enquanto que com o uso de um processo, esta função demorou apenas 11.007 segundos, aproximadamente, tendo uma duração de 0.0011 segundos por ciclo.

Para além dos dois testes referidos nesta Secção, foi também efetuado um teste, em cada tipo de implementação, para verificar se todas as atualizações do estado de uma partida de DOTA 2, escritas para ficheiro, eram lidas pela função `read_log()`. De modo a determinar que não havia falhas em qualquer uma das duas implementações, foi desenvolvido um pequeno algoritmo (Algoritmo 4), que imprime o número de falhas consecutivas de leitura do *logfile* da partida, no terminal. Enquanto a partida não terminar, são obtidas todas as *ticks*, bem como as *ticks* anteriores. Se a diferença entre uma nova *tick* e a anterior a essa for maior do que um, significa que houve falha de pelo menos uma leitura consecutiva de estado de uma partida.

Uma vez aplicado o algoritmo em cada implementação (com *thread* e com processo), foi possível verificar que nenhuma falha de leitura do estado do jogo ocorreu, em qualquer uma das duas implementações, pelo que se conclui que qualquer uma das duas implementações testadas pode ser utilizada, sem problemas, na obtenção de estados de partidas. É possível concluir que o uso de um processo para executar a função `read_log()` é mais viável do que utilizando uma *thread*, chegando a ser quase duas vezes mais rápido o uso de processos no caso do primeiro teste (0.0025/0.0013) e 1.7 vezes mais rápido no caso do segundo teste (0.0019/0.0011). Foi por isso que, devido a deste facto, aliado ao processamento paralelo apenas possibilitado por processos, em Python, que evitam totalmente falhas de leitura de *logs*, se decidiu implementar o módulo `PythonDota2` com o uso de processos em vez de *threads*.

Algoritmo 4: Algoritmo que imprime o número de falhas de leitura do *logfile* da partida.

```
Result: Imprimir número de falhas de leitura do logfile da partida.
tick = 0;
num_falhas = 0;
while partida_nao_terminou do
    tick_anterior = tick;
    tick = obter_tick();
    diff = tick - tick_anterior;
    if tick >0 then
        if diff >0 then
            num_falhas = num_falhas + 1;
            print(num_falhas);
        else
        end
    else
    end
end
```

4.2 Validação do Sistema Desenvolvido

De modo a verificar que o módulo criado (`PythonDota2`) permite a criação de um agente de DOTA 2, através da interação com um herói, foi desenvolvido um agente que demonstrasse que o herói controlado pudesse executar ações, tendo conhecimento do estado atual da partida.

O agente desenvolvido implementa a mecânica de jogo do DOTA 2, denominada *stacking* de *creeps* neutros (acumulação de *creeps* neutros), que permite a acumulação de *creeps* neutros em campos da *jungle* do mapa do jogo. Esta acumulação de *neutral creeps* começa quando um herói importuna estas unidades, levando a que elas o ataquem e, se o herói fugir delas, que elas o persigam. Para que esta mecânica de jogo seja bem sucedida, o herói tem de fugir das unidades, após as importunar, para o mais longe possível das mesmas, de modo a que elas o sigam e deixem o seu campo livre, voltando ao mesmo, segundos mais tarde. Esta mecânica tem de ser executada aproximadamente entre os 52 e 54 segundos de cada minuto, dependendo do campo de *creeps*, pois quando se der início a um novo minuto da partida, aparecem novos elementos neutros nos campos livres. Ora, ao aparecerem novos *creeps* num campo, é criada uma *stack* (acumulação), composta pelos antigos *creeps* neutros, que foram importunados, e os que apareceram no novo minuto. No caso específico desta política, a mecânica *stacking* é feita num campo de *neutrals* presente na *jungle* da equipa *Dire*.

O *script* de Python criado para efetuar esta mecânica utiliza apenas dois comandos, `ActionQueue_MoveDirectly(x, y)` e `ActionQueue_AttackUnit(handle, bOnce)`.

O primeiro comando é utilizado para fazer o agente navegar no mapa até à localização dada pelas coordenadas x e y da função, que se localiza perto do campo de *creeps* neutros, para se aproximar dos mesmos e para fugir deles. O segundo comando é necessário durante a noite, pois apenas é possível importunar os *creeps* atacando um deles, pois estão a dormir, através de um *handle* para uma das unidades do campo e de uma variável booleana que define se se pretende atacar uma ou mais vezes, sendo o seu valor `True` pois apenas se pretende atacar e fugir.

As funções que retornam variáveis do estado da partida fundamentais para a definição desta política são:

- `game_state()`, função que retorna um valor inteiro representativo do estado atual da partida, e.g. escolha de heróis, pré partida, fase de preparação, etc.;
- `time()`, função que retorna o tempo da partida, sob o tipo de dados `float`;
- `time_of_day()`, função que retorna a altura do dia na partida, sob o tipo de dados `float`;
- `neutrals()`, função que retorna as unidades neutras presentes numa área de 1600 unidades à volta do agente.

Utilizando estas funções é possível obter a informação pretendida para a execução de *stacking*, utilizando os comandos previamente descritos. Utiliza-se a função `game_state()` para obter o estado com valor 5, respetivo à fase de preparação do jogo. Seguidamente são extraídos os minutos e segundos da função `time()` para controlar o início de *stacking*, bem como o valor retornado pela função `time_of_day()`, para definir se é necessário (caso seja de noite) atacar um *neutral* ou não (caso seja de dia). Caso seja de noite, é utilizada a função `neutrals()` para verificar se existem *creeps* neutros na área do agente; se existirem é possível, então, atacar um dos *neutrals*. Uma vez importunados os *neutrals*, o agente foge para uma localização longe da do campo de *creeps* e o *stacking* é finalizado. Este processo é repetido em cada minuto, a partir do minuto 1 da partida. A lógica completa desta mecânica encontra-se na Listagem A.1 e o *script* LUA do herói a controlar encontra-se na Listagem A.2.

4.2.1 Demonstração do Funcionamento do Agente Desenvolvido

Para demonstrar o funcionamento do agente desenvolvido foi necessário alterar, ligeiramente, duas funções, `send_command()` e `read_log()`, do módulo `PythonDota2`. À primeira foi, simplesmente, adicionado um `print` para indicar, no *output* do terminal, quais são as *ticks* em que determinados comandos são **enviados**. À segunda foi, também, adicionado um `print`, para indicar, no *output* do terminal, qual é o ficheiro *log* para o qual é escrito o estado da partida em questão, bem como a indicação de que foram **recebidos** comandos em determinadas *ticks*.

A partida foi criada com 6.5 minutos de duração, (incluindo a fase de preparação de 1.5 minutos), no modo *All Pick*, para o herói *Ursa*, na equipa *Dire*. As imagens representativas da automatização de criação de partidas através da GUI do DOTA 2 encontra-se no Apêndice A desta dissertação.

Sabendo qual o *logfile* da partida em questão, é possível inspecionar o mesmo para verificar quando são recebidos os comandos. No caso específico deste teste, como demonstra a Listagem 4.1, o *logfile* a consultar é `console.4227553159.log`, de modo a verificar quando são executados os comandos enviados nas *ticks* representadas nesta Listagem.

Listagem 4.1: *Output* produzido no terminal, utilizando o *script* Python (ver A.1) e o *script* LUA para o herói (ver A.2).

```
console.4227553159.log
Command at tick 1
Command at tick 6060
Command at tick 6061
Command at tick 6062
Command at tick 7860
Command at tick 7861
Command at tick 7862
Command at tick 9660
Command at tick 9661
Command at tick 9662
Command at tick 11470
Command at tick 11471
Command at tick 11472
```

Listagem 4.2: *Output* produzido no *logfile* da partida, utilizando o *script* Python (ver A.1) e o *script* LUA para o herói (ver A.2).

```
[VScript] Queued move to (1000, 3950).
Cmd received at tick 2.
[VScript] Queued move to (1200, 3400).
Cmd received at tick 6061.
[VScript] Queued move to (500, 5000).
Cmd received at tick 6062.
[VScript] Queued move to (1000, 3950).
Cmd received at tick 6063.
[VScript] Queued move to (1200, 3400).
Cmd received at tick 7861.
[VScript] Queued move to (500, 5000).
Cmd received at tick 7862.
[VScript] Queued move to (1000, 3950).
Cmd received at tick 7863.
[VScript] Queued move to (1200, 3400).
```

```
Cmd received at tick 9661.
[VScript] Queued move to (500, 5000).
Cmd received at tick 9662.
[VScript] Queued move to (1000, 3950).
Cmd received at tick 9663.
[VScript] Queued attacking unit once.
Cmd received at tick 11471.
[VScript] Queued move to (500, 5000).
Cmd received at tick 11472.
[VScript] Queued move to (1000, 3950).
Cmd received at tick 11473.
```

Consultando o *logfile* referido e, comparando com o *output* produzido no terminal (ver 4.1), é possível concluir que todos os comandos foram enviados e respetivamente recebidos e executados, na *tick* seguinte ao envio dos comandos, acrescentando apenas um atraso, entre envio e execução de comandos, de 33 milissegundos, que é o tempo do processamento de uma *tick*. Os comandos enviados consecutivamente são também recebidos e executados consecutivamente. Assim, conclui-se que a implementação do agente foi bem sucedida, conseguindo efetuar a mecânica de jogo *stacking*.

Todas as imagens demonstrativas desta mecânica de jogo denominada *stacking*, bem como a sequência de ecrãs, resultante da automatização, para a criação de partidas, encontram-se no apêndice A, no final desta dissertação.

4.3 Teste de Consistência de Dados

É fundamental nesta *framework* existir consistência de dados, porque o utilizador tem de aceder à informação de partidas do DOTA 2 e codificar o seu agente conforme o que realmente está a acontecer na partida. Não pode existir qualquer tipo de incongruência na informação obtida. Isto significa que a informação, que o utilizador do módulo `PythonDota2` obtém através das variáveis da classe `State`, necessita de ser igual à informação presente no *logfile*, que contém o estado de uma partida a cada *tick*.

A obtenção de informação do estado de uma partida, presente num *logfile* é, como já referido anteriormente, feita através da função `read_log()`, que atribui os valores obtidos deste ficheiro a atributos da classe `State`. O uso de variáveis do estado do jogo, presentes nesta mesma classe, é feito através de um *script* Python que utilize as funções do módulo `PythonDota2`. Portanto, é possível comparar o estado do jogo que é obtido na função `read_log()` com a informação que o utilizador do módulo obtém.

Assim sendo, para testar a consistência de dados, foi construído um teste que consiste em, através da função `read_log()`, escrever 1000 linhas (correspondentes ao estado de uma partida nas primeiras 1000 *ticks*) para um novo ficheiro, chamado (`estado_modulo.txt`), enquanto que o *script* criado pelo utilizador também escreve para um outro ficheiro (`estado_utilizador.txt`) algumas variáveis representativas do estado do jogo, formando um estado para cada *tick*. Uma vez finalizada a partida e escritos os

	Ep. 1	Ep. 2	Ep. 3	Ep 4.	Ep 5.	Média
Total de estados lidos	1000	1000	1000	1000	1000	1000
Lidos pelo utilizador	985	986	985	992	971	984
%Lidos	98.5%	98.6%	98.5%	99.2%	97.1%	98.4%
Consistentes	917	906	924	908	905	912
%Consistência	93.1%	91.9%	93.8%	91.5%	93.2%	92.7%
Inconsistentes	68	80	61	84	66	72
%Inconsistência	6.9%	8.1%	6.2%	8.5%	6.8%	7.3%

Tabela 4.1: Resultados de 5 episódios de teste de consistência de dados.

dois novos ficheiros com informações sobre o estado da partida, são comparados através de um programa criado para este teste, que tratará de ler os dois ficheiros e compará-los, de modo a que seja possível concluir, ou não, que a informação, lida e obtida pelo utilizador e pela função `read_log()`, é consistente. O programa identifica quais são os estados, obtidos pelo utilizador do módulo, que não pertencem aos 1000 estados obtidos pela função `read_log()`. Assim é possível verificar se existe inconsistência, pois se, no primeiro conjunto de estados, existirem estados cuja informação não é inteiramente igual aos estados presentes no segundo conjunto é porque esses estados não são completamente consistentes.

Foi então criada uma partida de teste com 1000 *ticks* de duração, no modo *All Pick*, para o herói Ursa, na equipa *Dire* e aumentada a escala de tempo da partida ao máximo, com uso do comando `host_timescale 100` no terminal do DOTA 2, na automatização de criação de partidas através da GUI do jogo. Este teste foi repetido 5 vezes. Os resultados encontram-se na Tabela 4.1.

É possível verificar, analisando a Tabela 4.1, que, em média, foram obtidos 984 estados pelo utilizador, sendo 912, desses estados, coerentes com os estados reais e que existe, em média uma percentagem de consistência de dados de 92.7%.

Comparando o último estado inconsistente obtido do último episódio (Listagem 4.3), através do programa criado para comparar os dois tipos de obtenção de estados, é possível verificar que, comparando com excerto do ficheiro `estado_modulo.txt` (Listagem 4.4) obtido através deste episódio, o estado inconsistente é igual ao primeiro estado da Listagem 4.4, exceto o tempo da partida, que se encontra no segundo estado da Listagem 4.4.

Listagem 4.3: Estado obtido pelo utilizador, que é inconsistente. O estado lê-se da seguinte maneira: *posX*;*posY*;*hp*;*maxHp*;*mana*;*maxMana*;*time*.

```
1045.30188;3979.248291;632;632;267;267;-66.767021179199; ...
```

Listagem 4.4: Dois estados obtidos a partir do ficheiro estado_modulo.txt.

```
[VScript] ;Vector;0x35c958;[1045.301880;3979.248291;
384.000000];632;632;267;267;-66.800354003906 ...

[VScript] ;Vector;0x35d220;[1036.664795;3973.671875;
384.000000];632;632;267;267;-66.767021179199 ...
```

Ora, é possível verificar na Listagem 4.4 que existe inconsistência no estado obtido pelo utilizador, no entanto, a inconsistência existe apenas em uma variável do estado e a diferença é mínima pois, considerando que cada *tick* é processada em 33 milissegundos pelo DOTA 2, é quase desprezável uma diferença de um valor de uma variável de uma *tick* para outra, levando a que o estado inconsistente não seja tão problemático. Também se pode considerar que esta ligeira inconsistência não é grave quando comparada ao acesso a estados de partidas pelos agentes de DOTA 2 desenvolvidos pela OpenAI [10], que é apenas efetuado a cada quatro *ticks*, levando a uma diferença de tempo entre obtenção de estados de cerca de 133 milissegundos, enquanto que na nossa implementação se obtêm novos estados das partidas a cada 33 milissegundos.

As pequenas inconsistências de estados obtidos pelo utilizador devem-se à obtenção dos atributos presentes na classe `State`, através de *getters*, separadamente. Apesar da atualização destes atributos ser efetuada com recurso a um *mutex* presente na variável de condição utilizada na função `read_log()`, a obtenção dos mesmos, na perspetiva do utilizador é feita separadamente, através de funções como, por exemplo, `get_locX()` ou `get_hp()`. Ao obter separadamente cada uma destas variáveis, correspondentes a um estado de uma partida, existe a possibilidade de, na próxima obtenção de uma variável de um mesmo estado, esta já ter sido atualizada pela função `read_log()`, por ter sido lido um novo estado, originando um estado inconsistente obtido pelo utilizador, como ilustra a Listagem 4.3, que contem elementos de dois estados, representados na Listagem 4.4.

Uma forma de eliminar a possibilidade de obtenção de estados inconsistentes é implementar uma função no módulo `PythonDota2`, que o utilizador possa usar, para retornar um objeto `State` que contém as variáveis do estado de uma partida. Estas variáveis estarão atualizadas corretamente a cada nova *tick* obtida, sendo sempre consistente o estado obtido, pois é atualizado recorrendo a um *mutex* presente na função `read_log()`. Uma vez obtido o estado a cada nova *tick* obtida, o utilizador pode usar funções reescritas do módulo desenvolvido (*getters*), que retornem as variáveis presentes no objeto `State` obtido pelo utilizador.

Concluindo, existe uma pequena percentagem de estados não obtidos pelo utilizador, bem como uma pequena percentagem de estados inconsistentes. No entanto, a diferença entre estados inconsistentes obtidos e os estados reais é mínima, pois apenas varia entre duas *ticks* consecutivas, correspondendo a apenas 33 milissegundos de inconsistência de informação sobre parte do estado da partida.

4.4 Teste de Acesso a Variáveis do Estado de Partidas

Como é possível verificar, na Secção 4.3, o utilizador, ao aceder a variáveis do estado de partidas, através de um *script* que utilize funções do módulo `PythonDota2`, não consegue obter todos os estados de uma partida, conseguindo apenas, em média, obter 98.4% dos mesmos. Se extrapolarmos esta percentagem de obtenção de estados para uma partida que dure 5 minutos (sem contar com 1.5 minutos da fase de preparação), ou seja 300 segundos mais 90, obtemos, aproximadamente 11818 *ticks*, sabendo que uma *tick*, associada a uma

atualização de estado da partida, tem uma duração aproximada de 33 milissegundos, como demonstra a expressão seguinte.

$$390/0.033 \approx 11818 \tag{4.2}$$

Ora, 1.6% de 11818 é, aproximadamente, igual a 189 estados não lidos. No entanto, é necessária uma melhor compreensão sobre com que frequência e em que momentos estes estados não são obtidos e se a sua falta é crucial para uma partida de DOTA 2.

Assim, foi desenvolvido um teste que consiste na criação de várias partidas, de modo a observar em que momentos de cada partida existem falhas consecutivas na leitura de estados e se as falhas consistem de 1 ou mais estados que não são obtidos. Foram criadas 5 partidas, com o herói *Ursa*, na equipa *Dire*, no modo *All Pick*, com duração de 5 minutos (mais 1.5 minutos da fase de preparação) cada uma e obtido o estado de cada uma das partidas, através de funções disponibilizadas pelo módulo `PythonDota2`. Em todas estas partidas de teste houve um aumento da escala de tempo, para o valor máximo. Foram, também, registados os instantes, em cada partida, em segundos, em que se registavam falhas consecutivas na obtenção de um ou mais estados, bem como o número de falhas em cada um desses momentos. Este registo foi efetuado através do agente implementado em Python, que obtém o tempo da partida, bem como as *ticks* respetivas, em cada estado da partida, verificando-se se a diferença entre dois estados obtidos consecutivamente é maior do que um; se for, regista-se o número de falhas consecutivas a seguir a esse instante, bem como o tempo da partida em que houve a primeira falha de obtenção de estado. Os resultados obtidos encontram-se na Tabela 4.2 que contém, para além do número total de falhas de leitura em cada episódio testado, o tempo de execução e o *Speedup* de processamento calculado para cada episódio.

É possível concluir que, com base nos dados da Tabela 4.2, houve uma média de 73 estados não observados pelo utilizador, num total de cerca de 11818 estados obtidos (cerca de 0.618%) pela função `read_log()` do módulo `PythonDota2`, em cada um dos cinco episódios. A média de tempo de processamento dos testes é de 49.560 segundos, obtendo-se, assim um *Speedup* médio de 7.87 em relação aos 390 segundos de cada partida. O *Speedup*, aplicado no contexto deste teste, significa a aceleração obtida do tempo de execução em cada teste comparativamente ao tempo de cada partida, sendo o valor 1 de *Speedup* o valor que se obteria se não houvesse um aumento da escala de tempo das partidas. Os gráficos, resultantes de cada um dos episódios testados com o número de estados não lidos em determinados instantes de cada episódio, encontram-se no Apêndice A desta dissertação, mais concretamente nas Figuras A.15, A.16, A.17, A.18 e A.19. Cada um destes gráficos possui, no eixo dos XX, o tempo da partida (obtido através da função global `DotaTime()`, da API de *bots* de DOTA 2), de -90 a 0 segundos (fase de preparação) e de 0 a 300 segundos (decorrer do resto da partida) e, no eixo dos YY, o número de falhas de leitura consecutivas de estado a seguir a determinado instante de uma partida.

Conclui-se que, na média dos 5 episódios testados, cerca de 90% das falhas de leitura de estados acontecem na fase de preparação da partida, que não é uma fase muito relevante no total da partida. Estas falhas iniciais devem-se ao processamento inicial dos elementos do

Episódio	1	2	3	4	5	Média
Total de falhas de leitura	57	77	72	78	80	73
Tempo de execução (s)	48.756	48.950	50.238	49.930	49.925	49.560
Tempo total da partida (s)	390	390	390	390	390	390
Speedup	8	7.97	7.76	7.81	7.81	7.87

Tabela 4.2: Tabela com dados obtidos de cada episódio do teste realizado às falhas de leitura por partida.

mapa e da partida de DOTA 2. Observa-se também que a maior parte de estados seguidos que não são lidos, corresponde a apenas 1 *tick*, em determinados instantes. Em todos os episódios encontram-se poucos instantes de 2, 3 e 4 estados da partida consecutivos que não foram obtidos. Apenas no episódio 3 se observou que 5 estados da partida consecutivos não foram lidos em apenas dois instantes (ver A.17). Outra ilação que se pode tirar é que, com o decorrer do tempo das partidas, há cada vez menos falhas na obtenção de estados, sendo na sua maior parte, apenas 1 estado não obtido em cada um dos instantes mais tardios de cada um dos episódios. Assim, o número de falhas de leitura de estados consecutivos, torna-se linear, tendendo para que, com o decorrer da partida, passem a existir cada vez menos falhas na obtenção do estado do jogo.

Ainda assim, não é muito eficiente, em termos computacionais, numa *framework* como esta, existir a possibilidade de mais do que 2 estados consecutivos não serem obtidos pelo utilizador, apesar de apenas se perderem cerca de 100, 133 ou 166 milissegundos seguidos, não obtendo 3, 4, ou 5 estados consecutivos, respetivamente. Para além deste facto, o utilizador, ao usar funções do módulo desenvolvido, ciclicamente, utilizará ciclos de processador em vão, pois vários ciclos irão iterar sobre o mesmo estado obtido.

Devido aos dois factos enunciados no parágrafo anterior, foi implementada uma função no módulo `PythonDota2`, chamada `wait_for_tick()` para, quando chamada pelo utilizador, esperar por um novo estado do jogo, levando a que não sejam desperdiçados ciclos de processamento. Esta função criada, utiliza a função `wait()` da classe Condition [52], que espera até ser notificada pelo processo que está a usar o *mutex* para atualizar o estado da partida; uma vez notificada pelo processo responsável pela leitura de *logfiles* e atualização da classe `State`, a espera termina.

Foi efetuado um novo teste, idêntico ao teste já descrito nesta Secção da dissertação, mas desta vez, com a adição da função `wait_for_tick()` ao início de cada ciclo. Todos os detalhes da criação de partidas, bem como o número de partidas criadas para este teste são iguais aos do teste referido anteriormente. Os resultados obtidos deste novo teste encontram-se na Tabela 4.3. A Tabela 4.3 contém, para além do número total de falhas de leitura em cada episódio testado, o tempo de execução e o *Speedup* de processamento calculado para cada episódio.

Com base nos dados da Tabela 4.3, conclui-se que houve uma média de 218 estados não observados pelo utilizador, num total de cerca de 11818 estados obtidos (cerca de 1.845%) pela função `read_log()` do módulo `PythonDota2`, em cada um dos cinco episódios, ou seja,

Episódio	1	2	3	4	5	Média
Total de falhas de leitura	234	226	206	210	214	218
Tempo de execução (s)	40.567	42.620	42.807	42.361	43.232	42.318
Tempo total da partida (s)	390	390	390	390	390	390
Speedup	9.61	9.15	9.11	9.21	9.02	9.22

Tabela 4.3: Tabela com dados obtidos de cada episódio do teste realizado às falhas de leitura por partida, com a utilização da função `wait_for_tick()`.

aproximadamente 3 vezes mais estados não obtidos, em média, do que no teste anterior. A média de tempo de processamento dos testes é de 42.318 segundos, obtendo-se, assim um *Speedup* médio de 9.22 em relação aos 390 segundos de cada partida, ou seja, cerca de 1.17 (9.22/7.87) vezes mais rápido que o *Speedup* do teste anterior. Os gráficos, resultantes de cada um dos episódios testados com o número de estados não lidos em determinados instantes de cada episódio, com o uso da função `wait_for_tick()`, encontram-se no Apêndice A desta dissertação, mais concretamente nas Figuras A.20, A.21, A.22, A.23 e A.24. Cada um destes gráficos possui, no eixo dos XX, o tempo da partida (obtido através da função global `DotaTime()`, da API de *bots* de DOTA 2), de -90 a 0 segundos (fase de preparação) e de 0 a 300 segundos (decorrer do resto da partida) e, no eixo dos YY, o número de falhas de leitura de estado em determinado instante.

Conclui-se que, na média dos 5 episódios testados, cerca de 95% das falhas de leitura de estados acontecem na fase de preparação da partida (mais 5% do que no teste sem o uso da função `wait_for_tick()`, que não é uma fase muito relevante no total da partida. Observa-se também que, nos episódios 1 e 2, apenas 1 estado à vez, em determinados instantes, de cada uma destas partidas, não é lido. Nos restantes episódios, acontece o mesmo, exceto em um momento, no episódio 3 e 4, em que não são lidos 2 estados consecutivos e em dois momentos no episódio 5 em que não são lidos também 2 estados consecutivos. Outra ilação que se pode tirar é que, à semelhança do teste anterior, com o decorrer do tempo das partidas, há cada vez menos falhas na obtenção de estados. Com esta implementação, conclui-se que, com o decorrer da partida, passem a existir cada vez menos falhas na obtenção do estado do jogo.

	Sem `wait_for_tick()`	Com `wait_for_tick()`
% de perda total de estados	0.618%	1.845%
% de falhas na fase de preparação	90%	95%
Perda máx. de estados consecutivos	5 (66 ms)	2 (166 ms)
Moda de perda estados consecutivos	1 (33 ms)	1 (33 ms)
Evita ciclos redundantes?	Não	Sim

Tabela 4.4: Tabela comparativa entre testes realizados, com e sem o uso da função `wait_for_tick()`, na obtenção de estados de partidas de DOTA 2, através do módulo `PythonDota2`.

Comparando os dois testes feitos (Tabela 4.4), é possível afirmar que a obtenção de estados de partidas com o uso da função `wait_for_tick()` é preferível, pois, apesar de existirem mais falhas neste segundo teste, no total, da leitura do estado do jogo, do que no primeiro teste realizado (cerca de 3 vezes mais, em média), as falhas são, maioritariamente, isoladas. Isto quer dizer que, em determinados instantes de uma partida de DOTA 2, apenas se perdem 33 milissegundos (perda de 1 estado) da partida e, em raras ocasiões, perdem-se, no máximo, 66 milissegundos (perda de 2 estados consecutivos) da partida, enquanto que não utilizando esta função é possível perder, no máximo, 166 milissegundos (perda de 5 estados consecutivos). Uma maior percentagem de falhas de obtenção de estados na fase de preparação de partidas de DOTA 2 também contribui para a preferência do uso da função `wait_for_tick()` na obtenção de estados, pois esta fase é a menos importante numa partida. Outro aspeto relevante foi o aumento de *Speedup* médio de processamento de partidas de 7.87 para 9.22, sem o uso da função `wait_for_tick()` e com o uso da mesma, respetivamente. Por fim, uma importante vantagem do uso desta função é a capacidade de evitar a execução de ciclos de processamento redundantes.

4.5 Conclusão

Neste Capítulo foram efetuados três testes de desempenho da *framework* desenvolvida para agentes de DOTA 2 através de uma API em Python. Um dos testes consiste numa comparação de desempenho entre duas implementações de uma função de obtenção de estados de partidas de DOTA 2 através de *logfiles*. O segundo teste analisa a consistência da obtenção de estados de partidas deste jogo. O último teste de desempenho examina o acesso a estados de partidas, em que foram comparadas duas implementações ligeiramente distintas. Para além destes testes demonstrou-se que é possível utilizar a *framework* criada para desenvolver um agente de DOTA 2.

Capítulo 5

Conclusões e Trabalho Futuro

Neste Capítulo é feita uma análise e reflexão do trabalho desenvolvido, durante a escrita desta dissertação, como um todo. Este Capítulo encontra-se dividido em 3 Secções: comparação do trabalho desenvolvido com outras implementações de interfaces para desenvolvimento de agentes de DOTA 2 (Secção 5.1), uma reflexão sobre os testes realizados, fornecendo, assim, uma ideia geral dos aspetos positivos do trabalho desenvolvido (Secção 5.2), bem como o que há a melhorar em trabalho futuro (Secção 5.3).

5.1 Comparação da *Framework* Desenvolvida com outras Implementações Existentes

De forma a avaliar a pertinência da nossa implementação, é importante compará-la com as implementações estudadas. Nesta Subsecção pretende-se comparar a nossa implementação de *framework* para desenvolvimento de agentes de DOTA 2 com outras implementações existentes, nomeadamente:

1. *Dotabots-ml-tools* [31] (Subsecção 2.4.1);
2. *CreepBlockAI* [33] (Subsecção 2.4.2);
3. *Dota2Comm* [34] (Subsecção 2.4.3);
4. *Dota2-WebAI* [35] (Subsecção 2.4.4);
5. *Dota2AIFramework* [36] (Subsecção 2.4.5);
6. *Dota2DQN* [37] (Subsecção 2.4.5);
7. *Dota2-FullOverwrite* [38] (Subsecção 2.4.5).

A Tabela 5.1 estabelece a comparação entre as várias implementações estudadas, referidas anteriormente, e a implementação de uma *framework* para o desenvolvimento de

agentes de DOTA 2, desenvolvida nesta tese, denominada `PythonDota2`. A comparação é feita com base em várias características, nomeadamente o modo de comunicação entre a *framework* e o jogo e se é efetuado o envio de comandos e receção do estado de partidas.

Implementação/ Característica	1	2	3	4	5	6	7	`PythonDota2`
Compatível com DOTA 2?	Não	Não	Sim	Não	Não	Sim	N/A	Sim
Usa a API *Workshop Tools*?	Sim	Sim	Não	Sim	Sim	Não	Não	Não
Modo de comunicação	*Webserver* (W)	W	Ficheiros (F)	W	W	F	N/A	F
Linguagem de desenvolvimento	Python	Python	Python	Python	Java	Python	LUA	Python
Envio de comandos?	Não	Não	Sim	Sim	Sim	Sim	N/A	Sim
Módulo para desenvolvimento?	Não	Não	Sim	Sim	Sim	Não	N/A	Sim
Recebe estado de partidas?	Sim	Sim	Não	Sim	Sim	N/A	Sim	Sim
Finalizada ou em desenvolvimento?	Não	Não	Não	Não	Não	Não	Não	Sim

Tabela 5.1: Comparação entre implementações existentes de interfaces entre o DOTA 2 e ambientes de programação e a *framework* desenvolvida nesta tese.

Analisando a Tabela 5.1, é possível verificar que apenas três implementações estudadas poderiam ser utilizadas para o desenvolvimento de agentes, devido a apenas utilizarem a API de *scripting* de *bots* de DOTA 2 e não utilizarem a API DOTA 2 *Workshop Tools*. Apesar de utilizar apenas a API de *bots*, a implementação 7 é apenas uma reestruturação do modo de programação de *bots* em LUA.

A implementação 3 efetua a comunicação com o jogo através de escrita e leitura de ficheiros; todas as outras implementações, à exceção da 7 (em que a comunicação é intrínseca), utilizam um *webserver* que processa pedidos `HTTP` para efetuar a comunicação entre o jogo e as respetivas *frameworks*.

A linguagem de desenvolvimento nestas *frameworks* é maioritariamente Python, com a exceção da implementação 5 (Java) e 7 (LUA). A implementação 3 fornece um meio de envio de mensagens para o DOTA 2, mas não recebe mensagens do jogo, que poderiam ser utilizadas para a leitura do estado de partidas.

A implementação 6 utiliza o meio de comunicação da implementação 3, no entanto, não é claro que receba o estado de partidas, devido à falta de informação no repositório [37]. Todas as outras implementações recebem o estado de partidas e efetuam envio de comandos para o DOTA 2, com a exceção da implementação 1 e 2, que processam o estado do jogo a atualizam um modelo com base em algoritmos de ML, através de um *webservice*, modelo esse que é interpretado pelo DOTA 2 e, conforme certos parâmetros, certas ações são despoletadas, sem ser necessário a receção de comandos.

As implementações 1, 2 e 6 têm uma codificação que não permite o seu uso de uma

maneira minimamente simples (sem ter de ser editada pelo utilizador), para além destas implementações já conterem algoritmos de ML, que não contribuem para a clareza das *frameworks* desenvolvidas.

Atualmente, nenhuma das implementações estudadas se encontra em desenvolvimento ou finalizada.

A implementação de uma *framework* para desenvolvimento de agentes de DOTA 2, realizada nesta tese não utiliza a API *Workshop Tools* de DOTA 2, utiliza ficheiros como método de comunicação com o jogo, foi desenvolvida em Python, permite o envio de comandos para os agentes, bem como da receção dos estados das partidas e fornece um módulo Python de fácil utilização para qualquer utilizador interessado neste tema poder utilizar.

Assim, é possível concluir que a *framework* desenvolvida para implementação de agentes de DOTA 2 é a mais completa, quando comparada com as outras implementações estudadas e reúne todas as condições necessárias para uma utilização fácil. Tanto quanto sabemos, em relação às outras *frameworks* estudadas, a ferramenta desenvolvida nesta tese é a única que permite a implementação de agentes de DOTA 2, devido à implementação de um módulo simples de utilizar, com as funções necessárias para este tipo de implementação, mantendo a robustez necessária e obrigatória num sistema deste género.

5.2 Análise de Testes e Demonstração da *Framework* Desenvolvida

Com base nos testes e demonstração efetuados à *framework* desenvolvida, detalhados no Capítulo 4, é possível tecer algumas conclusões importantes sobre a eficiência da *framework* desenvolvida.

O uso de um processo para a execução paralela, a um processo principal que instancie a classe `PythonDota2`, da função que trata de obter os estados das partidas, é preferível ao uso de uma *thread* para o mesmo efeito. Isto deve-se ao facto de, ao utilizar a linguagem de programação Python, o uso de *threads* não ser o mais eficiente para paralelismo, como explicado no Capítulo 3. Isto comprova-se com o primeiro teste efetuado, presente na Secção 5.1, em que o uso de um processo é cerca de duas vezes mais rápido do que o uso de uma *thread*, para executar a função `read_log()`.

A política *hard-coded* representativa da mecânica de jogo denominada *stacking* é efetuada com sucesso, utilizando o módulo `PythonDota2`. Nesta demonstração, cada comando enviado em determinada *tick*, a partir do módulo, é sempre executado no *script* LUA do herói exatamente na *tick* seguinte, respetivamente. É possível verificar as ações que o agente toma, analisando as Figuras A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.7, presentes no Apêndice desta dissertação.

Existe uma consistência média de 92.7% dos estados acessados pelo utilizador, estes que são, em média 98.4% dos estados obtidos através da função `read_log()`. Este valor é aceitável porque as inconsistências de dados são parciais num mesmo estado obtido

inconsistente, em comparação ao estado seguinte, em que a diferença entre estados é de apenas 33 milissegundos.

A função `wait_for_tick()` é responsável por esperar por uma atualização de estado. Esta pode ser utilizada pelo utilizador do módulo desenvolvido, para limitar o número de ciclos de processamento na execução de um programa que recorra ao módulo `PythonDota2`. Com a utilização da função `wait_for_tick()`, a obtenção de estados é mais eficiente em relação à obtenção de estados sem o uso desta função, apesar de existirem mais falhas de obtenção de estados, em média, Isto deve-se ao facto de o valor máximo de estados não obtidos consecutivos ser de apenas 1, enquanto que, não utilizando a função, o mesmo valor é de 5. Sabendo que uma atualização de estado de partidasno DOTA 2 dura cerca de 33 milissegundos, é preferível perder apenas 1 atualização de estado (perda de 33 milissegundos de estado da partida), em determinado instante, do que perder 5 atualizações consecutivas (perda de 166 milissegundos de estado da partida). Ainda assim, a perda de 1 atualização do estado do jogo não é quase irrelevante, quando comparado às implementações da OpenAI, de agentes de DOTA 2, em que a obtenção de estados é feita a cada quarta *tick* [14]. Para além deste facto, 95% das falhas de obtenção de estado, utilizando a função `wait_for_tick()` estão presentes na fase de preparação do jogo de DOTA 2, que é pouco relevante numa partida.

Os testes realizados foram efetuados com aceleração de partidas ao máximo, através do comando `host_timescale 100.0`, utilizado no terminal do DOTA 2, na automatização de criação dos ambientes de partidas, obtendo-se resultados bastante eficientes, com um *Speedup* médio de 9.22 (utilizando a função `wait_for_tick()`), do tempo total das partidas em relação ao tempo de execução dos testes. Se a máquina, em que estes testes foram executados, tivesse maior poder de computação, os resultados obtidos seriam, potencialmente, ainda melhores.

5.3 Trabalho Futuro

Como trabalho futuro, existem várias possibilidades de melhoria desta *framework* desenvolvida.

Relativamente ao número de ações que um agente pode tomar, que se encontra algo limitado, apenas será necessário acrescentar funções ao módulo `PythonDota2`, que sejam representativas de ações que um agente de DOTA 2 possa tomar, com base na API de *scripting* de *bots* de DOTA 2 [12]. De igual forma, é necessário acrescentar ao *script* do agente um reconhecimento dos novos comandos implementados no módulo em Python.

No que diz respeito ao número de variáveis do estado das partidas, também podem ser adicionados elementos, mas é necessário ter em atenção que o desempenho do *script* LUA pode ser afetado devido a grande quantidade *output* e processamento de informação. De maneira a ser possível obter mais variáveis do estado de partidas, através do módulo desenvolvido é necessário acrescentar à função `read_log()` um *parsing* adequado aos novos tipos de variáveis impressas nos *logfiles* das partidas.

É possível melhorar o processamento do estado de partidas, de modo a torná-lo mais

eficiente, ao utilizar memória partilhada entre processos, em vez de um *manager* de processos. No entanto a memória partilhada em Python não possibilita a partilha de estruturas de dados de Python como as listas ou os dicionários, o que pode tornar a mudança de implementação um pouco drástica.

Outro tipo de melhoria a fazer a esta *framework* é possibilitar a implementação multiagente, ou seja, treinar vários agentes, simultaneamente, com a possibilidade de se defrontarem mutuamente nas mesmas partidas, pois é possível ter vários heróis numa mesma partida de DOTA 2, desde que sejam todos diferentes. É necessário que sejam todos diferentes pois ao utilizar o *script* LUA respetivo a um herói, não existe possibilidade de 2 heróis iguais efetuarem diferentes ações, com o uso de *scripts*, pois cada herói tem apenas um *script* atribuído a si próprio.

Por fim, é possível acrescentar funcionalidades que o utilizador do módulo desenvolvido possa utilizar, como registar o fator de aceleração de partidas que pretende, se pretende desenvolver variados agentes ou mesmo se pretende jogar com ou contra o seu próprio agente nas partidas de treino e/ou teste.

5.4 Conclusão

Em suma, nesta tese, foi criada uma ferramenta de *software*, simples de utilizar, que permite um começo rápido de implementação de agentes de DOTA 2. Com esta *framework*, já é possível criar algoritmos de ML para desenvolver agentes de DOTA 2, através do uso do módulo `PythonDota2` e de um *script* LUA para o herói que se pretenda desenvolver, que pode ser obtido através do *template* presente no Apêndice A desta dissertação.

Qualquer alteração a esta *framework* pode ser feita com alguma facilidade, devido a ser composta por um módulo desenvolvido em Python, que contém funções que podem ser alteradas separadamente sem causar muita instabilidade no sistema. O *script* modelo para os heróis respetivos também pode ser alterado de forma a completar quaisquer alterações efetuadas no módulo `PythonDota2`.

Bibliografia

[1] Claude E Shannon. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[3] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[5] Starcraft II. `https://starcraft2.com/en-us/game`. Acedido pela última vez: 24 de Setembro de 2018.

[6] Deepmind and Blizzard open starcraft II as an AI research environment. `https://deepmind.com/blog/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment/`. Acedido pela última vez: 25 de Setembro de 2018.

[7] DOTA 2. `https://http://www.dota2.com/play/`. Acedido pela última vez: 25 de Setembro de 2018.

[8] Dota 2 OpenAI. `https://blog.openai.com/dota-2/`. Acedido pela última vez: 25 de Setembro de 2018.

[9] More on Dota 2. `https://blog.openai.com/more-on-dota-2//`. Acedido pela última vez: 25 de Setembro de 2018.

[10] OpenAI. Openai Five. `https://blog.openai.com/openai-five/`, 2018. Acedido pela última vez: 25 de Setembro de 2018.

[11] J. M. Font Fernandez and T. Mahlmann. The dota 2 bot competition. *IEEE Transactions on Games*, pages 1–1, 2018.

[12] Dota Bot Scripting. https://developer.valvesoftware.com/wiki/Dota_Bot_Scripting. Acedido pela última vez: 25 de Setembro de 2018.

[13] The International. http://www.dota2.com/international/overview/. Acedido: 10 de Novembro de 2018.

[14] Dota 2 OpenAI Five. https://blog.openai.com/openai-five/. Acedido pela última vez: 25 de Setembro de 2018.

[15] About OpenAI. https://openai.com/about/. Acedido pela última vez: 9 de Novembro de 2018.

[16] Dota 2 Workshop Tools/Scripting/API. https://developer.valvesoftware.com/wiki/Dota_2_Workshop_Tools/Scripting/API. Acedido pela última vez: 29 de Setembro de 2018.

[17] Rogelio Adobbati, Andrew N Marshall, Andrew Scholer, Sheila Tejada, Gal A Kaminka, Steven Schaffer, and Chris Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the second international workshop on Infrastructure for Agents, MAS, and Scalable MAS*, volume 5, page 6. Montreal, Canada, 2001.

[18] John E Laird. It knows what you're going to do: adding anticipation to a Quakebot. In *Proceedings of the fifth international conference on Autonomous agents*, pages 385–392. ACM, 2001.

[19] Dota 2 Minimap. https://dota2.gamepedia.com/File:Minimap_7.07.pngs. Acedido: 14 de Outubro de 2018.

[20] Dota 2 Vision. https://dota2.gamepedia.com/Vision. Acedido: 14 de Outubro de 2018.

[21] Dota 2 Roshan. https://dota2.gamepedia.com/Roshan. Acedido: 11 de Outubro de 2018.

[22] Dota 2 Attributes. https://dota2.gamepedia.com/Attributes. Acedido: 13 de Outubro de 2018.

[23] Dota 2 Passive Abilities. https://liquipedia.net/dota2/Passive. Acedido: 11 de Outubro de 2018.

[24] Dota 2 Talents. https://dota2.gamepedia.com/Talents. Acedido: 12 de Outubro de 2018.

[25] Dota 2 Buyback. https://dota2.gamepedia.com/Gold#Buyback. Acedido: 14 de Outubro de 2018.

[26] Introducing OpenAI. https://blog.openai.com/introducing-openai/. Acedido pela última vez: 8 de Novembro de 2018.

[27] Openai's Gym. http://gym.openai.com/. Acedido pela última vez: 9 de Novembro de 2018.

[28] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[29] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[30] Dota 2 Matchmaking Stats: MMR. https://dota.rgp.io/mmr/. Acedido pela última vez: 5 de Dezembro de 2018.

[31] dotabots-ml-tools Python. https://github.com/ThePianoDentist/dotabots-ml-tools. Acedido pela última vez: 8 de Outubro de 2018.

[32] Dota 2 Workshop Tools DLC. https://steamdb.info/app/313250/. Acedido pela última vez: 30 de Setembro de 2018.

[33] CreepBlockAI. https://github.com/Nostrademous/CreepBlockAI. Acedido pela última vez: 9 de Outubro de 2018.

[34] Dota2Comm. https://github.com/Keithenneu/dota2comm. Acedido pela última vez: 9 de Outubro de 2018.

[35] Dota2-WebAI. https://github.com/Nostrademous/Dota2-WebAI. Acedido pela última vez: 9 de Outubro de 2018.

[36] Dota2 AI Framework. https://github.com/lightbringer/dota2ai. Acedido pela última vez: 24 de Novembro de 2018.

[37] Dota2DQN. https://github.com/lenLRX/Dota2DQN. Acedido pela última vez: 9 de Outubro de 2018.

[38] Dota2 Full Overwrite. https://github.com/Nostrademous/Dota2-FullOverwrite. Acedido pela última vez: 24 de Novembro de 2018.

[39] Unreal Tournament: Game of the Year Edition. https://store.steampowered.com/app/13240/Unreal_Tournament_Game_of_the_Year_Edition/. Acedido pela última vez: 4 de Outubro de 2018.

[40] About Lua. https://www.lua.org/about.html. Acedido: 30 de Setembro de 2018.

[41] Quake II. https://store.steampowered.com/app/2320/QUAKE_II/. Acedido pela última vez: 4 de Outubro de 2018.

[42] John E Laird, Allen Newell, and Paul S Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

[43] George A Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[44] Lua's I/O Library. `https://www.lua.org/pil/21.html`. Acedido: 15 de Outubro de 2018.

[45] Dota 2 Ursa. `https://dota2.gamepedia.com/Ursa`. Acedido: 16 de Outubro de 2018.

[46] Steam. `https://store.steampowered.com/`. Acedido: 17 de Outubro de 2018.

[47] Compilation, Execution, and Errors: loadfile Function. `http://www.lua.org/pil/8.html`. Acedido: 18 de Outubro de 2018.

[48] Lua 5.1 Reference Manual: tonumber Function. `https://www.lua.org/manual/5.1/manual.html#pdf-tonumber`. Acedido: 18 de Outubro de 2018.

[49] Python 3 Threading Module. `https://docs.python.org/3/library/threading.html`. Acedido: 16 de Outubro de 2018.

[50] Python 3 Multiprocessing Module. `https://docs.python.org/3.4/library/multiprocessing.html`. Acedido: 16 de Outubro de 2018.

[51] Python 3 Threading Condition Objects. `https://docs.python.org/3/library/threading.html#condition-objects`. Acedido: 16 de Outubro de 2018.

[52] Python 3 Multiprocessing Condition Objects. `https://docs.python.org/3.4/library/multiprocessing.html?highlight=process#multiprocessing.Condition`. Acedido: 16 de Outubro de 2018.

[53] Tail. `http://man7.org/linux/man-pages/man1/tail.1.html`. Acedido: 16 de Outubro de 2018.

[54] PyAutoGUI. `https://pyautogui.readthedocs.io/en/latest/`. Acedido: 17 de Outubro de 2018.

[55] Xdotool: fake keyboard/mouse input, window management, and more. `https://github.com/jordansissel/xdotool`. Acedido: 26 de Novembro de 2018.

[56] Python 3 Subprocess Management. `https://docs.python.org/3/library/subprocess.html`. Acedido: 17 de Outubro de 2018.

[57] Python. `https://www.python.org/`. Acedido: 23 de Setembro de 2018.

[58] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[59] Cpython, the Python programming language. `https://github.com/python/cpython`. Acedido: 19 de Outubro de 2018.

[60] Python GIL. `https://wiki.python.org/moin/GlobalInterpreterLock`. Acedido: 20 de Outubro de 2018.

[61] Python Docs GIL. `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`. Acedido: 20 de Outubro de 2018.

[62] Sharing State Between Processes. `https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes`. Acedido: 30 de Outubro de 2018.

[63] Python Lists. `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`. Acedido: 31 de Outubro de 2018.

[64] Python Dictionaries. `https://docs.python.org/3/tutorial/datastructures.html#dictionaries`. Acedido: 31 de Outubro de 2018.

[65] Python BaseManager. `https://docs.python.org/2/library/multiprocessing.html#multiprocessing.managers.BaseManager`. Acedido: 31 de Outubro de 2018.

[66] Python BaseManager's Function Register. `https://docs.python.org/2/library/multiprocessing.html#multiprocessing.managers.BaseManager.register`. Acedido: 31 de Outubro de 2018.

[67] List of Console Commands: host_timescale. `https://dota2.gamepedia.com/List_of_Console_Commands#H`. Acedido: 22 de Novembro de 2018.

Apêndices

Apêndice A

Testes e Demonstrações

Listagem A.1: *Stacking* de *creeps* neutros.

```
from PythonDota2 import *

import math
import time


n_episodes = 1
bot = PythonDota2("dire", "ursa",
"templar assassin", "all pick", n_episodes)

for i in range(n_episodes):

    bot.start_dota() # iniciar DOTA~2
    bot.start_env_lobby() # iniciar o ambiente da partida

    not_done = True
    not_waiting = True
    not_stacked = True
    not_attacked = True

    while bot.game_state() != 4:
        # esperar pelo inicio da fase de preparacao da partida
        pass

    while not bot.dire_win() and not bot.radiant_win() \
    and bot.time() < 300:
        # partida acaba quando o tempo de jogo chegar
        # aos 300 segundos
```

```
    seconds, minutes = math.modf(bot.time()/60)
    seconds = seconds * 60

    if not_waiting:
        bot.ActionQueue_MoveDirectly(1000, 3950)
        not_waiting = False

    if 50 < seconds < 51:
        not_stacked = True
        not_attacked = True

    if seconds > 52.2 and minutes > 0 and not_stacked \
    and 0.25 < bot.time_of_day() < 0.75: # dia
        bot.ActionQueue_MoveDirectly(1200, 3400)
        # aproximacao aos creeps
        bot.ActionQueue_MoveDirectly(500, 5000)
        # fugir
        bot.ActionQueue_MoveDirectly(1000, 3950)
        # voltar a posicao inicial
        not_stacked = False

    elif seconds > 52.5 and minutes > 0 and not_stacked \
    and not_attacked and (bot.time_of_day() > 0.75 \
    or bot.time_of_day() < 0.25): # noite
        if bot.neutrals():
            for neutral in bot.neutrals():
                handle = neutral[0]
                if not_attacked:
                    bot.ActionQueue_AttackUnit(handle, True)
                    # acertar no creep mais perto uma vez
                    bot.ActionQueue_MoveDirectly(500, 5000)
                    # fugir
                    bot.ActionQueue_MoveDirectly(1000, 3950)
                    # voltar a posicao inicial
                    not_stacked = False
                    not_attacked = False
                    break

bot.set_game_over(True) # registar fim da partida

bot.close_env_lobby() # terminar o ambiente da partida
```

Listagem A.2: *Script* LUA utilizado no *stacking* de *creeps* neutros.

```
count = 1
tick = 0
function Think()
    local bot = GetBot();
    local loc = bot:GetLocation();
    local loc = tostring(loc);
    local loc = loc:gsub(" ", ";")

    local mana = tostring(bot:GetMana());
    local maxMana = tostring(bot:GetMaxMana());
    local hp = tostring(bot:GetHealth());
    local maxHp = tostring(bot:GetMaxHealth());
    local queuedActions = tostring(bot:NumQueuedActions())
    local neutrals_str = ""
    local lane_creeps_str = ""
    local heroes_str = ""
    local buildings_str = ""

    local nrby_ntrl_creeps = bot:GetNearbyNeutralCreeps(1600)
    for _, unit in pairs(nrby_ntrl_creeps) do
        neutrals_str = neutrals_str ..
        "[ " .. tostring(unit) .. " " ..
        unit:GetUnitName() .. " " .. --name
        unit:GetHealth() .. " " .. --HP
        unit:GetMaxHealth() .. " " .. --MaxHP
        unit:GetMana() .. " " .. --Mana
        unit:GetMaxMana() .. " " .. --MaxMana
        unit:GetAttackDamage() ..
        " " .. --AttackDamage with bonuses
        unit:GetArmor() .. " " .. --Armor
        GetUnitToUnitDistance(unit, bot) .. "] " --Distance
    end

    local nrby_ln_creeps = bot:GetNearbyLaneCreeps(1600, true)
    for _, unit in pairs(nrby_lane_creeps) do
        lane_creeps_str = lane_creeps_str ..
        "[ " .. tostring(unit) .. " " ..
        unit:GetUnitName() .. " " .. --name
        unit:GetHealth() .. " " .. --HP
        unit:GetMaxHealth() .. " " .. --MaxHP
        unit:GetMana() .. " " .. --Mana
```

```
    unit:GetMaxMana() .. " " .. --MaxMana
    unit:GetAttackDamage() ..
    " " .. --AttackDamage with bonuses
    unit:GetArmor() .. " " .. --Armor
    GetUnitToUnitDistance(unit, bot) .. "] " --Distance
end

local enemy_heroes = GetUnitList(UNIT_LIST_ENEMY_HEROES)
for _, unit in pairs(enemy_heroes) do
    heroes_str = heroes_str ..
    "[ " .. tostring(unit) .. " " ..
    unit:GetUnitName() .. " " .. --name
    unit:GetHealth() .. " " .. --HP
    unit:GetMaxHealth() .. " " .. --MaxHP
    unit:GetMana() .. " " .. --Mana
    unit:GetMaxMana() .. " " .. --MaxMana
    unit:GetAttackDamage() ..
    " " .. --AttackDamage with bonuses
    unit:GetArmor() .. " " .. --Armor
    GetUnitToUnitDistance(unit, bot) .. "] " --Distance
end

local nmy_buildings = GetUnitList(UNIT_LIST_ENEMY_BUILDINGS)
for _, unit in pairs(nmy_buildings) do
    unit_name = unit:GetUnitName()
    if unit_name == "npc_dota_goodguys_tower1_mid" then
        buildings_str = buildings_str ..
        "[ " .. tostring(unit) .. " " ..
        unit:GetUnitName() .. " " .. --name
        unit:GetHealth() .. " " .. --HP
        unit:GetMaxHealth() .. " " .. --MaxHP
        unit:GetMana() .. " " .. --Mana
        unit:GetMaxMana() .. " " .. --MaxMana
        unit:GetAttackDamage() ..
        " " .. --AttackDamage with bonuses
        unit:GetArmor() .. " " .. --Armor
        GetUnitToUnitDistance(unit, bot) .. "] " --Distance
    end
end

local state = ";" .. loc ..
    ";" .. hp ..
    ";" .. maxHp ..
```

```
    ";" .. mana ..
    ";" .. maxMana ..
    ";" .. DotaTime() ..
    ";" .. queuedActions ..
    ";" .. "[" .. neutrals_str .. "]" ..
    ";" .. GetTimeOfDay() ..
    ";" .. "[" .. lane_creeps_str .. "]" ..
    ";" .. "[" .. heroes_str .. "]" ..
    ";" .. "[" .. buildings_str .. "]" ..
    ";" .. tick ..
    ";" .. GetGameState()

print(state)
tick = tick +1;

local cmd =
    loadfile(GetScriptDirectory().."/tmp/"..count);
if cmd == nil then
    return
end

if string.startswith(cmd(), "Action_MoveDirectly") then
    cmd_split = split(cmd());
    x = tonumber(cmd_split[2]);
    y = tonumber(cmd_split[3]);

    bot:Action_MoveDirectly(Vector(x, y, 0.0));

elseif string.startswith(cmd(), "ActionPush_MoveDirectly")
        then
    cmd_split = split(cmd());
    x = tonumber(cmd_split[2]);
    y = tonumber(cmd_split[3]);

    bot:ActionPush_MoveDirectly(Vector(x, y, 0.0));

elseif string.startswith(cmd(), "ActionQueue_MoveDirectly")
        then
    cmd_split = split(cmd());
    x = tonumber(cmd_split[2]);
    y = tonumber(cmd_split[3]);
    to_print = "Queued move to " ..
        "(" .. x .. ", " .. y .. "). " ..
```

```
        "Cmd received at tick " .. tick-1 .. "."
    print(to_print)

    bot:ActionQueue_MoveDirectly(Vector(x, y, 0.0));

elseif string.startswith(cmd(), "Action_AttackUnit") then
    cmd_split = split(cmd());
    handle = cmd_split[2] .. " " .. cmd_split[3];
    once = cmd_split[4];
    for _, unit in pairs(nearby_neutral_creeps) do
        if tostring(handle) == tostring(unit) then
            if once == "True" then
                bot:Action_AttackUnit(unit, true);
            elseif once == "False" then
                bot:Action_AttackUnit(unit, false);
            end
        end
    end

elseif string.startswith(cmd(),"ActionPush_AttackUnit") then
    cmd_split = split(cmd());
    handle = cmd_split[2] .. " " .. cmd_split[3];
    once = cmd_split[4];
    for _, unit in pairs(nearby_neutral_creeps) do
        if tostring(handle) == tostring(unit) then
            if once == "True" then
                bot:ActionPush_AttackUnit(unit, true);
            elseif once == "False" then
                bot:ActionPush_AttackUnit(unit, false);
            end
        end
    end


elseif string.startswth(cmd(),"ActionQueue_AttackUnit") then
    cmd_split = split(cmd());
    handle = cmd_split[2] .. " " .. cmd_split[3];
    once = cmd_split[4];
    for _, unit in pairs(nearby_neutral_creeps) do
        if tostring(handle) == tostring(unit) then
            if once == "True" then
                bot:ActionQueue_AttackUnit(unit, true);
                to_print = "Queued attacking unit once. " ..
```

```
                        "Command received at tick " ..
                        tick-1 .. "."
                elseif once == "False" then
                    bot:ActionQueue_AttackUnit(unit, false);
                end
            end
        end

    elseif string.startswith(cmd(), "Action_ClearActions") then
        cmd_split = split(cmd());
        stop = cmd_split[2]
        if stop == "True" then
            bot:Action_ClearActions(true);
        elseif stop == "False" then
            bot:Action_ClearActions(false);
        end

    elseif string.startswith(cmd(), "Action_Delay") then
        cmd_split = split(cmd());
        delay = tonumber(cmd_split[2])
        bot:Action_Delay(delay)

    elseif string.startswith(cmd(), "ActionPush_Delay") then
        cmd_split = split(cmd());
        delay = tonumber(cmd_split[2])
        bot:ActionPush_Delay(delay)

    elseif string.startswith(cmd(), "ActionQueue_Delay") then
        cmd_split = split(cmd());
        delay = tonumber(cmd_split[2])
        bot:ActionQueue_Delay(delay)

    elseif cmd() == "hello" then
        print("Hi there");

    else
        print("");
    end

    count = count + 1;

end
```

```
function string.startswith(String,Start)
    return string.sub(String,1,string.len(Start)) == Start
end


function split(s, pattern, maxsplit)
    local pattern = pattern or " "
    local maxsplit = maxsplit or -1
    local s = s
    local t = {}
    local patsz = #pattern
    while maxsplit ~= 0 do
        local curpos = 1
        local found = string.find(s, pattern)
        if found ~= nil then
            table.insert(t, string.sub(s, curpos, found - 1))
            curpos = found + patsz
            s = string.sub(s, curpos)
        else
            table.insert(t, string.sub(s, curpos))
            break
        end
        maxsplit = maxsplit - 1
        if maxsplit == 0 then
            table.insert(t, string.sub(s, curpos - patsz - 1))
        end
    end
    return t
end
```

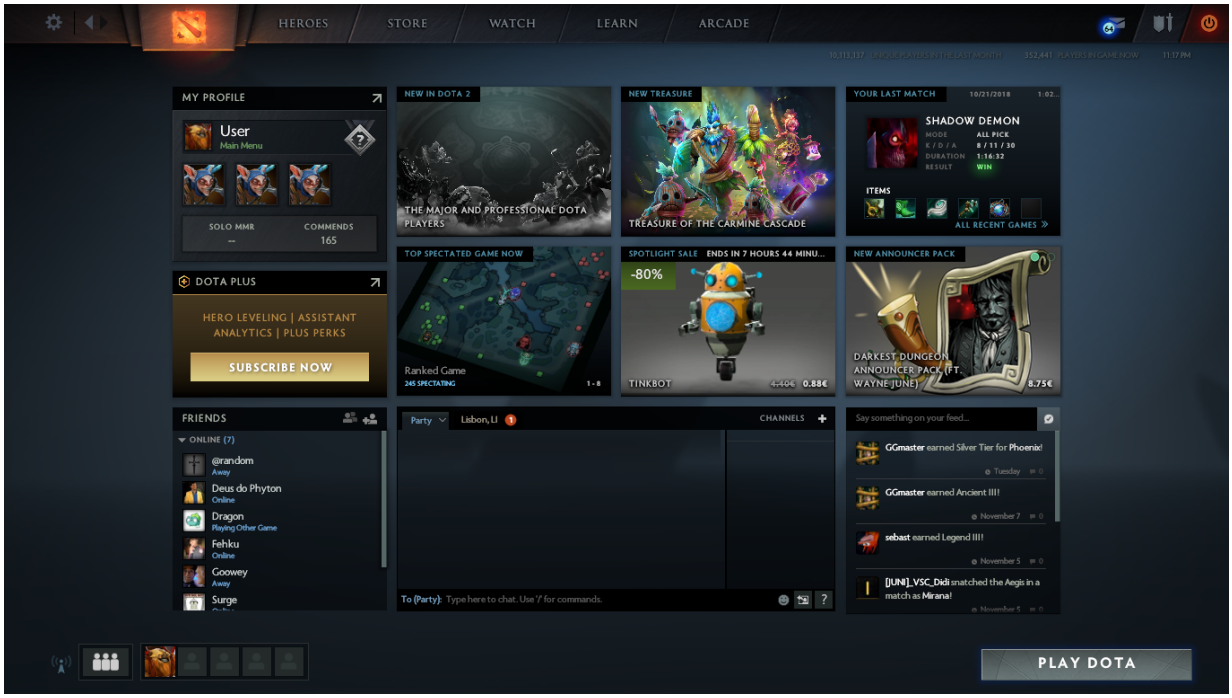


Figura A.1: Menu principal do DOTA 2.

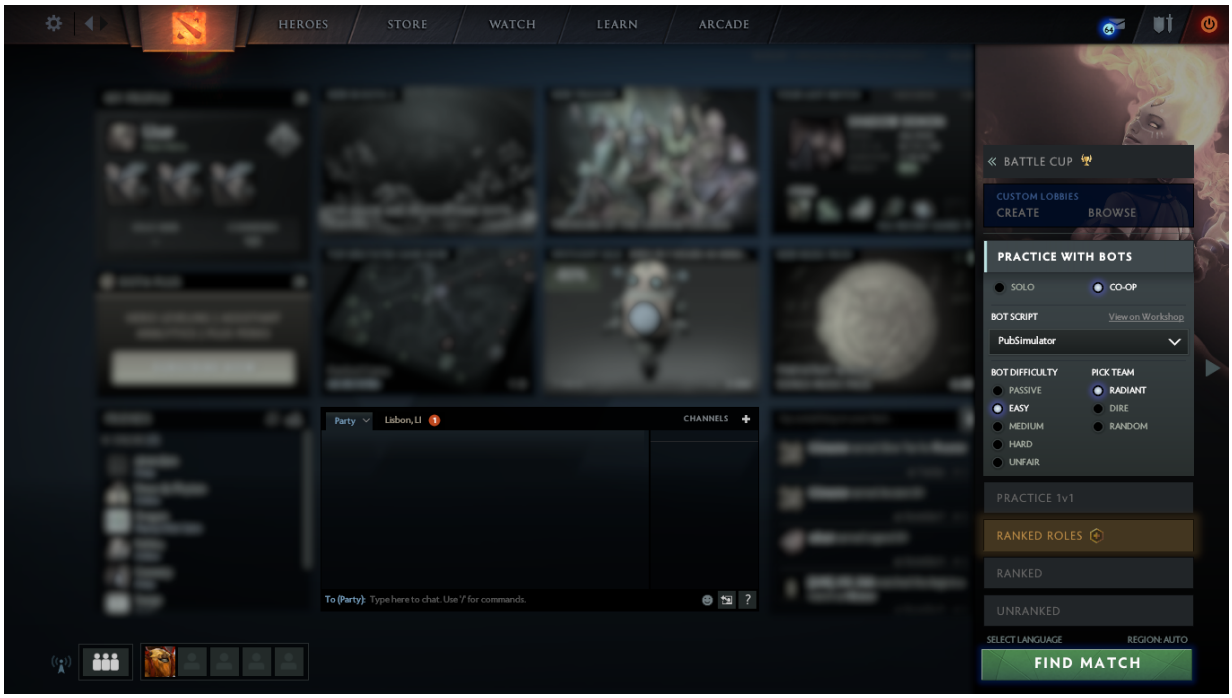


Figura A.2: Início de criação de uma partida de DOTA 2.

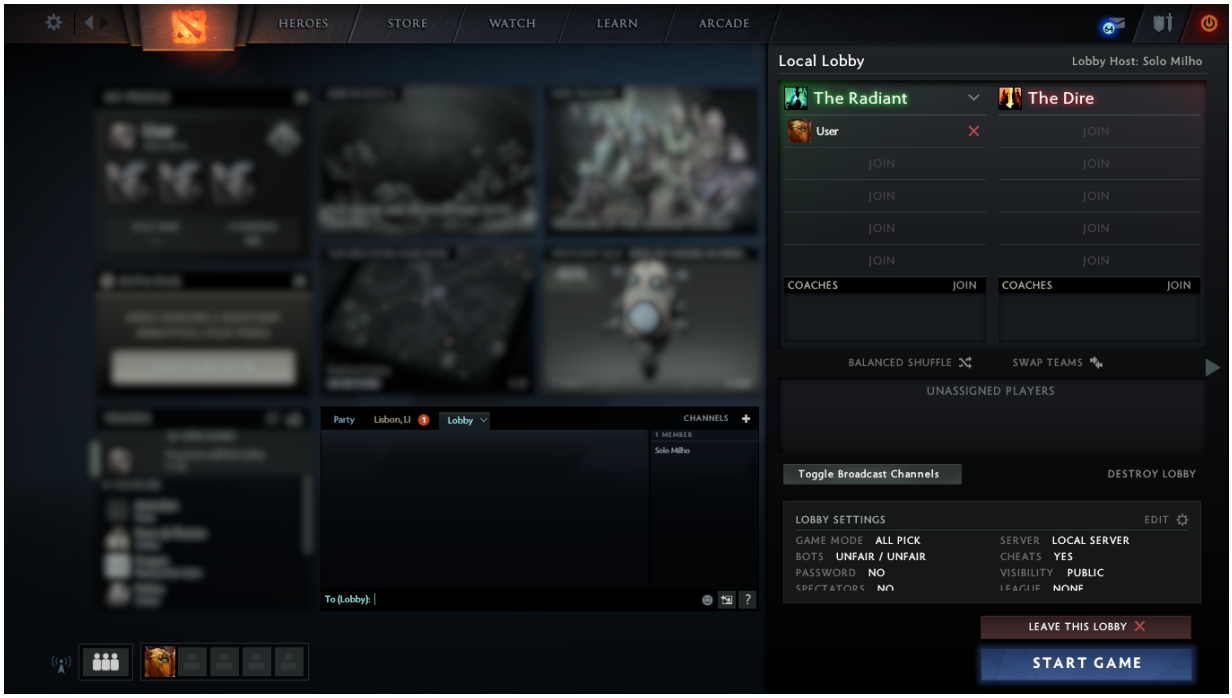


Figura A.3: Menu respetivo à criação de *lobby* local para uma nova partida de uma DOTA 2.

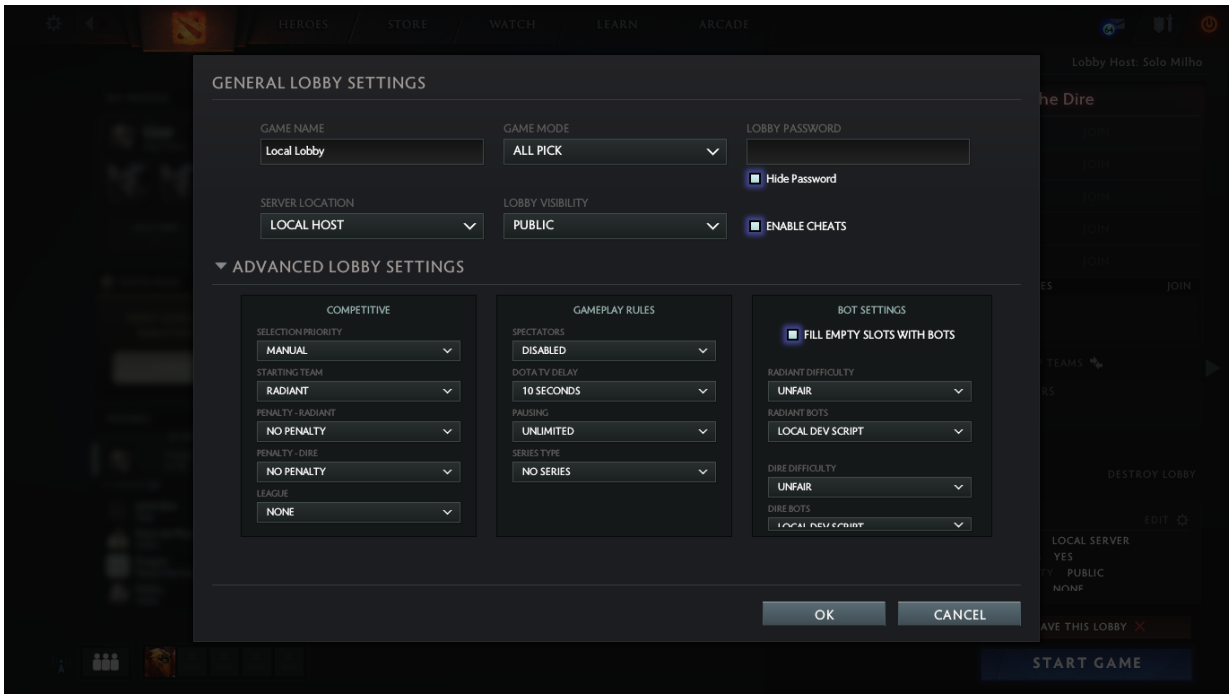


Figura A.4: Menu respetivo às definições para a criação de *lobby* local para uma nova partida de DOTA 2.

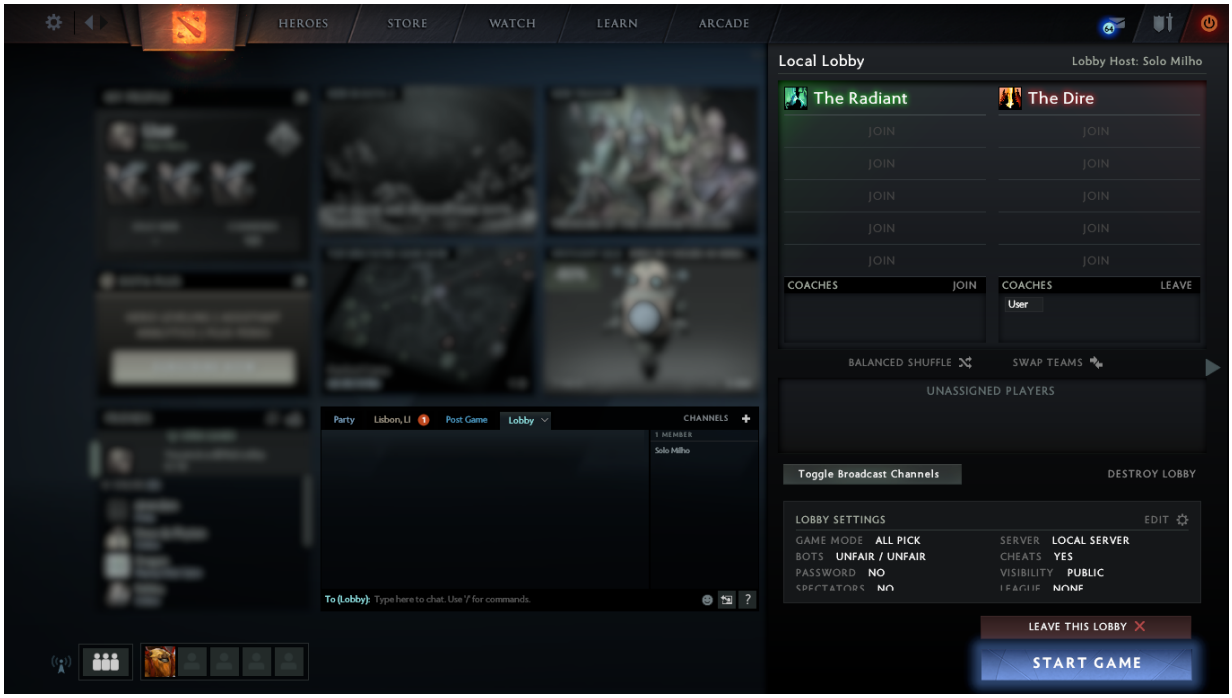


Figura A.5: Escolha de equipa para o utilizador, no modo *coach*, para permitir a visualização do agente de DOTA 2, sem ser possível controlá-lo.

Figura A.6: Aumento da escaça de tempo da partida ao máximo através do comando `host_timescale 100.0`..

Figura A.7: Início da fase de preparação da partida, com os heróis aliados ao herói *Ursa*, controlado pelo agente, presentes na base da respetiva equipa.

Figura A.8: Início do movimento do agente para perto da localização de um campo de *creeps* neutros, de modo a iniciar a mecânica de jogo denominada *stacking*.

Figura A.9: Chegada do agente à localização próxima de um campo de *creeps* neutros.

Figura A.10: Início do movimento do agente para perto dos *creeps* neutros, de modo a importuná-los.

Figura A.11: *Creeps* neutros são importunados e seguem o agente.

Figura A.12: *Creeps* neutros deixam de seguir o agente e retornam para o seu campo.

Figura A.13: *Creeps* neutros a retornarem ao seu campo.

Figura A.14: Mecânica de jogo denominada *stacking* efetuada, devido à existência de um novo conjunto de *creeps* neutros no campo.

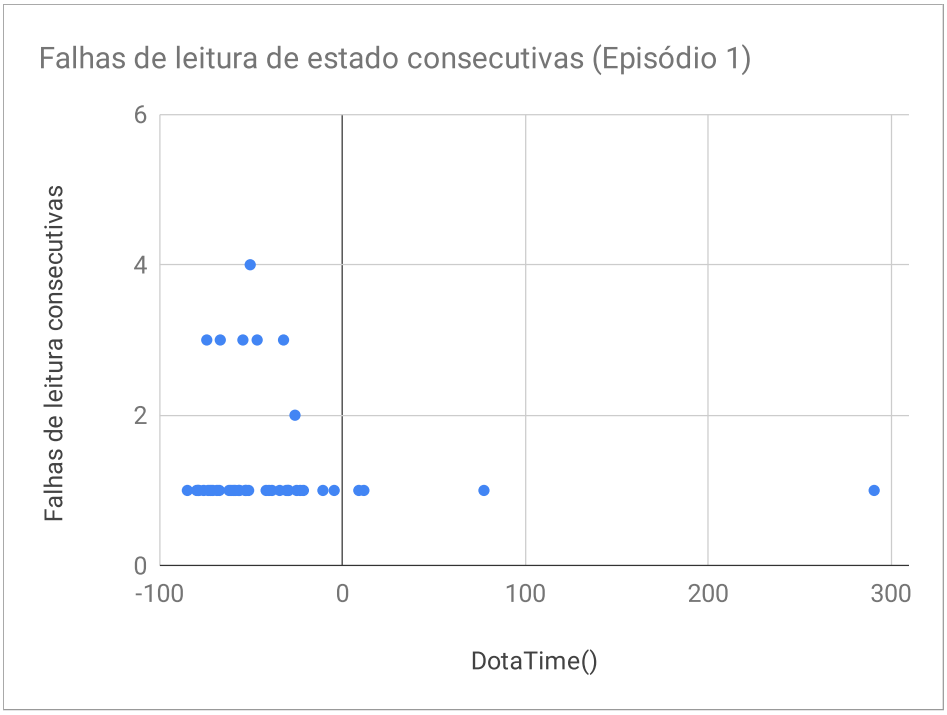


Figura A.15: Gráfico representativo do episódio 1 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida.

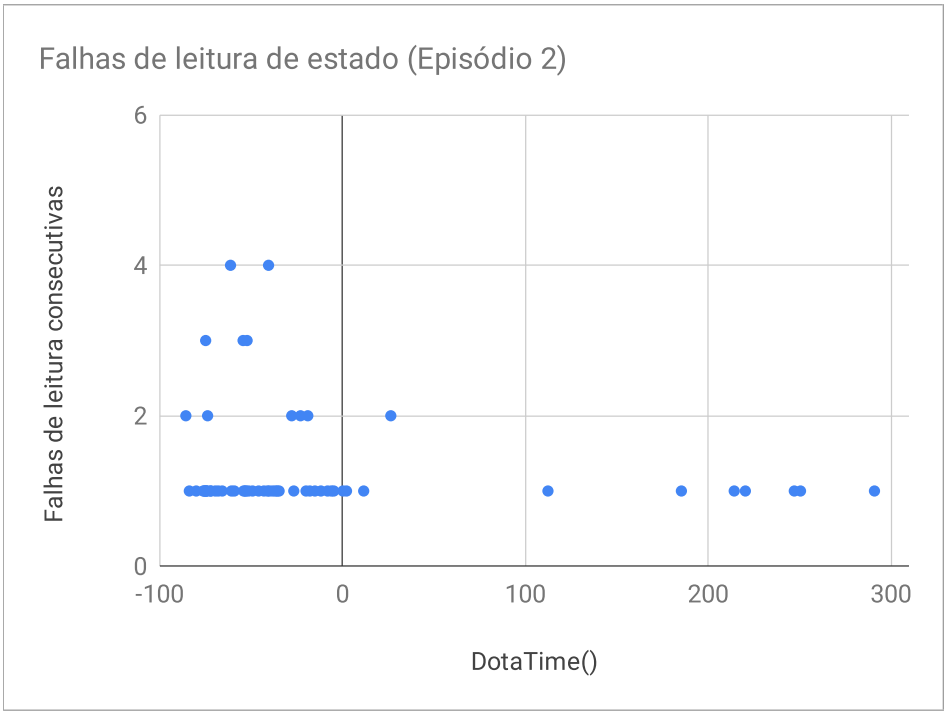


Figura A.16: Gráfico representativo do episódio 2 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida.

Figura A.17: Gráfico representativo do episódio 3 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida.

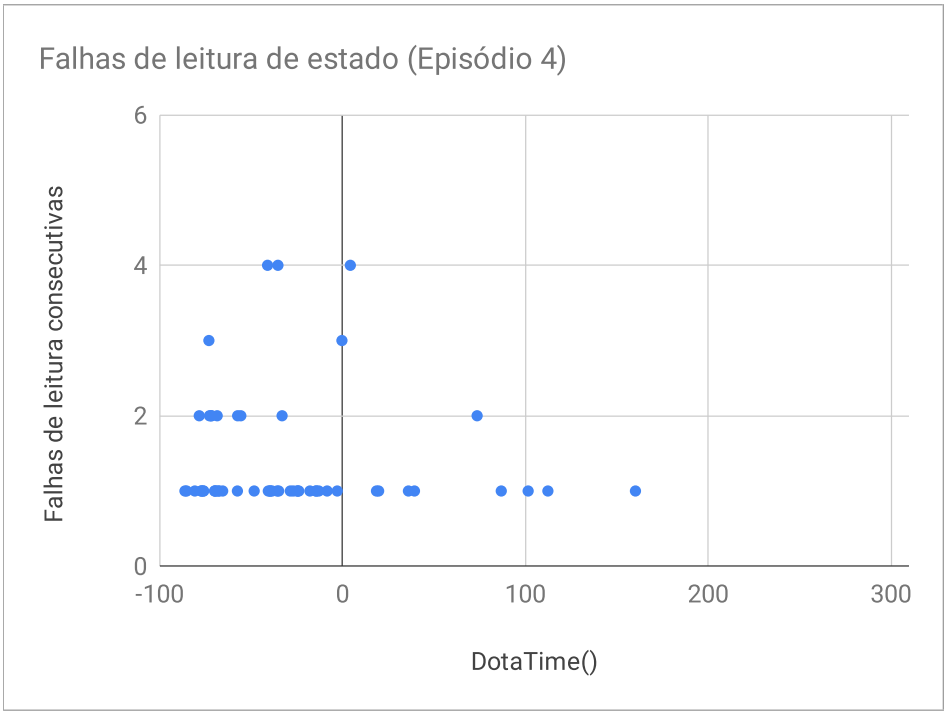


Figura A.18: Gráfico representativo do episódio 4 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida.

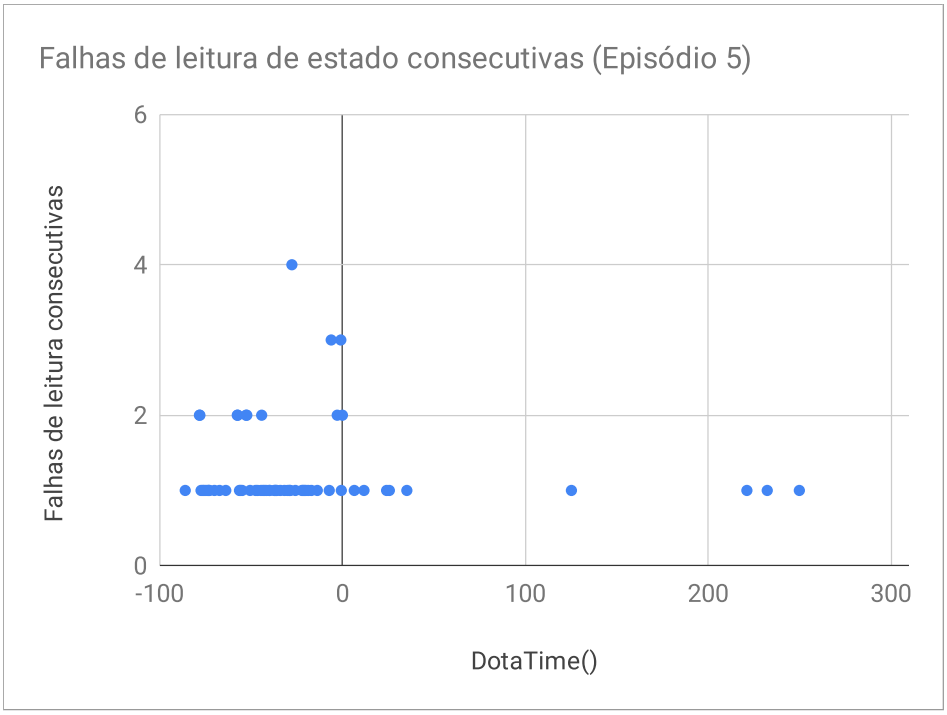


Figura A.19: Gráfico representativo do episódio 5 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida.

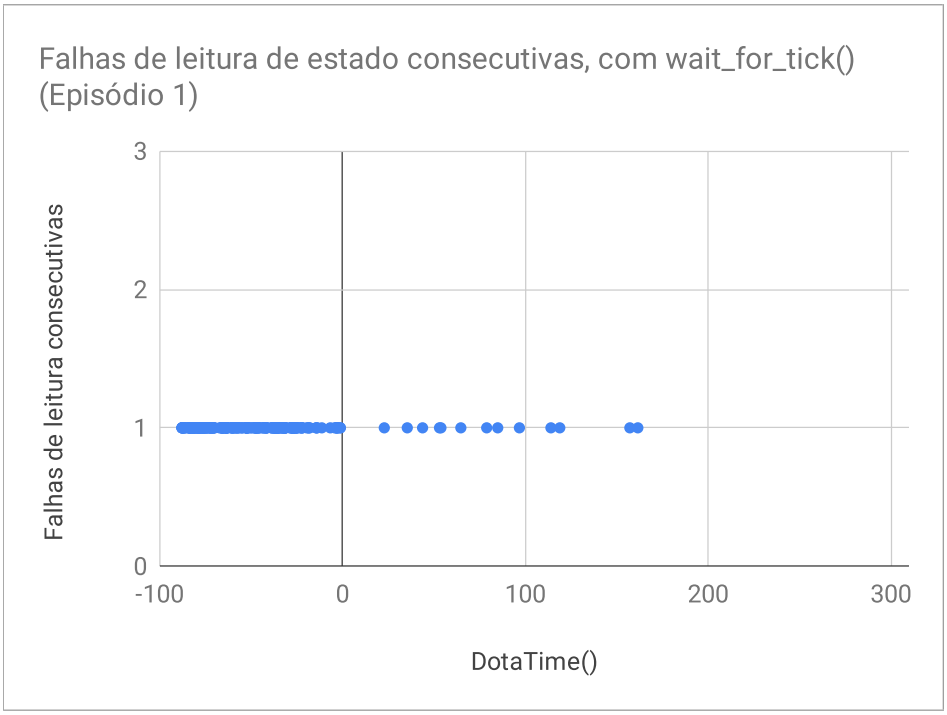


Figura A.20: Gráfico representativo do episódio 1 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida, com a função `wait_for_tick()`.

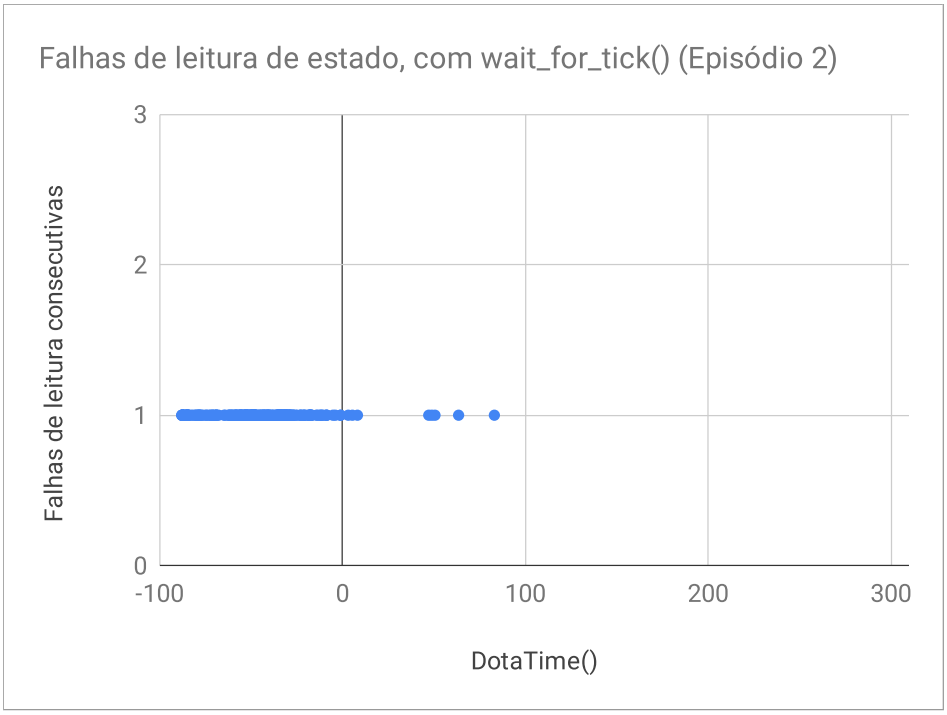


Figura A.21: Gráfico representativo do episódio 2 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida, utilizando a função `wait_for_tick()`.

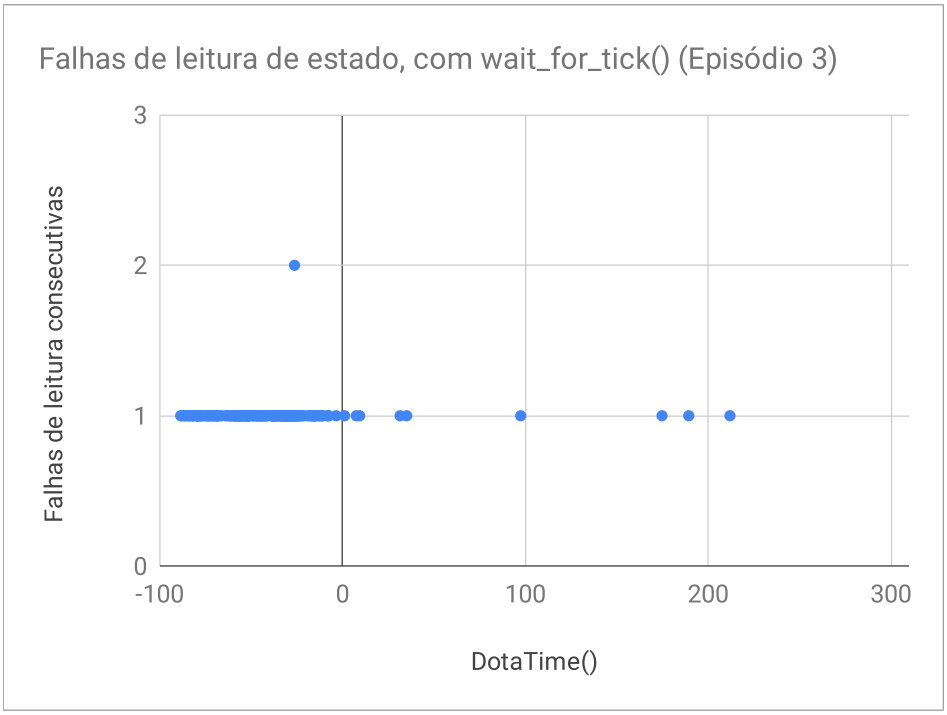


Figura A.22: Gráfico representativo do episódio 3 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida, utilizando a função `wait_for_tick()`.

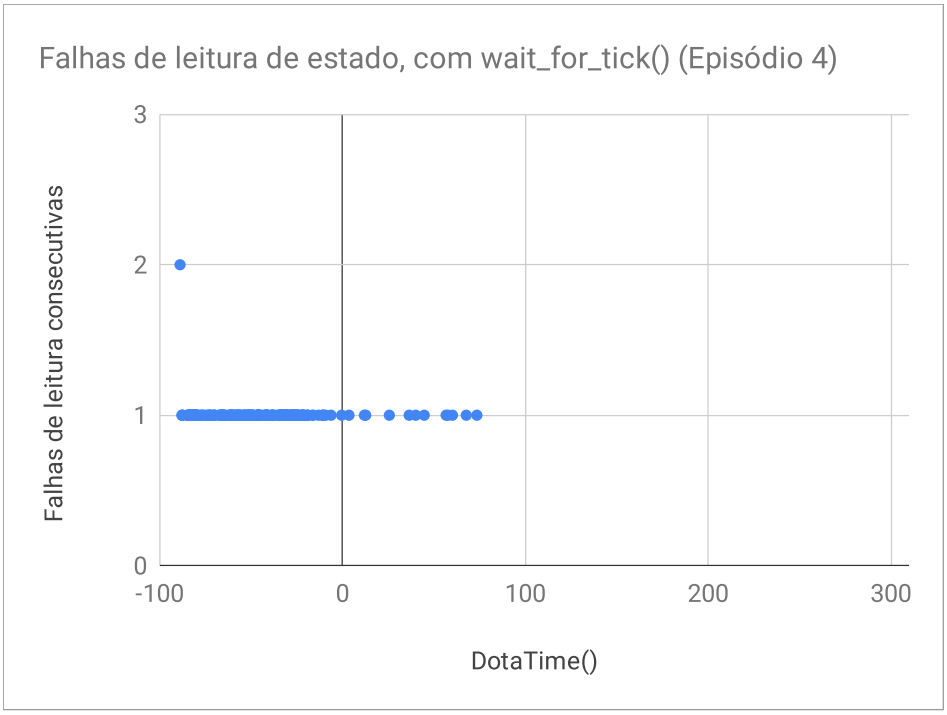


Figura A.23: Gráfico representativo do episódio 4 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida, utilizando a função `wait_for_tick()`.

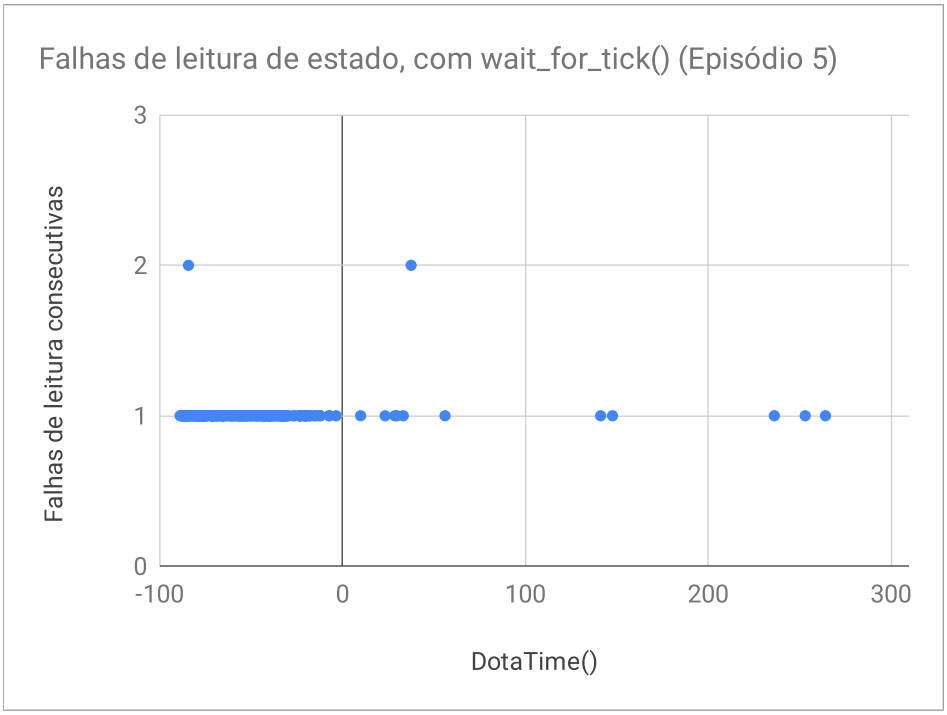


Figura A.24: Gráfico representativo do episódio 5 do teste do número de falhas de leitura consecutivas de estado a seguir a determinado tempo da partida, utilizando a função `wait_for_tick()`.